SAT Solvers: Theory and Practice

Clark Barrett

barrett@cs.nyu.edu

New York University

Formal Verification

- “[Formal] software verification . . . has been the Holy Grail of computer science for many decades”
  - Bill Gates [Gat02]
- Formal verification techniques can, in theory, prove beyond a doubt that a system is implemented correctly.
- In practice, there are still many challenges, but there are also success stories, and the technology is getting better.

Steps of Formal Verification

1. *Modeling*: Create a mathematical model of the system
   - A modeling error can introduce false bugs or mask real bugs
   - For many systems, this step can be done automatically
2. *Specification*: The properties which the system should satisfy must be stated in a formal language
   - Challenge to translate informal specifications into formal ones
   - Many languages: UML, CTL, PSL, Spec#, etc.
3. *Proof*: Prove that the model satisfies the specification
   - Better than testing: covers *all* cases
   - ...when it succeeds: *this is the hard part*

Automatic Theorem Provers

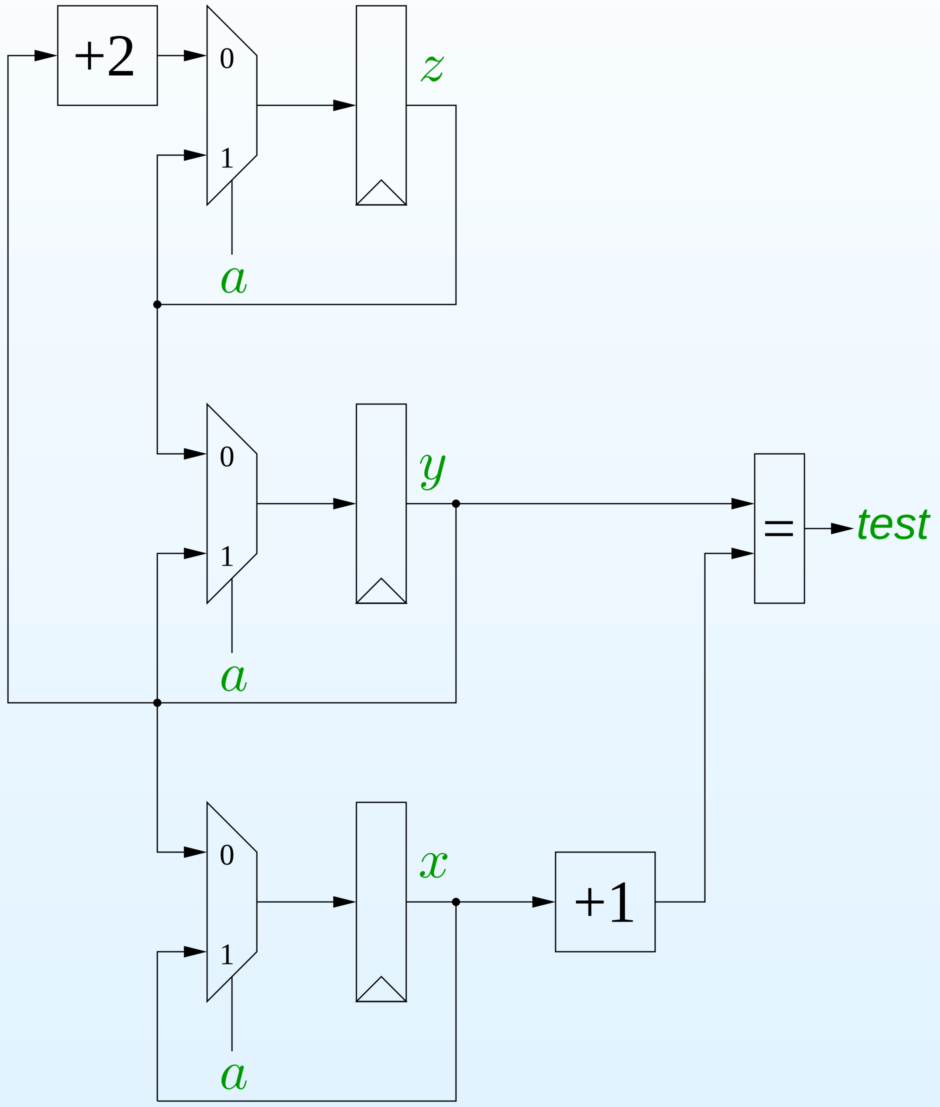
Many real-world verification efforts require human expertise to complete the proofs

If a computer can do the proof *automatically*, this greatly improves the feasibility of formal verification

Automatic theorem provers have improved significantly in recent years, enabling formal verification of larger and more complex systems

In these lectures, we will look at two techniques for automated theorem proving: *SAT solvers* and *SMT solvers*.

Circuit Example



Circuit Example

In this example, the value of *test* is always supposed to be *True*.

Circuit Example

In this example, the value of *test* is always supposed to be *True*.

Under what conditions does this hold?

Circuit Example

In this example, the value of *test* is always supposed to be *True*.

Under what conditions does this hold?

How do we prove it?

Circuit Example

In this example, the value of *test* is always supposed to be *True*.

Under what conditions does this hold?

How do we prove it?

We will come back to this question.

Roadmap

Boolean Satisfiability

- Propositional Logic
- Solving SAT
- Modeling for SAT

The Language of SAT solvers: Propositional Logic

A *SAT solver* solves the *Boolean satisfiabiliy* problem.

In order to understand the satisfiability problem, we must first define the language in which the problem is phrased.

The language is *propositional logic* [End00].

What is Logic?

A formal logic is defined by its *syntax* and *semantics*.

Syntax

- An *alphabet* is a set of symbols.
- A finite sequence of these symbols is called an *expression*.
- A set of rules defines the *well-formed* expressions.

Semantics

- Gives meaning to well-formed expressions
- Formal notions of induction and recursion are required to provide a rigorous semantics.

Propositional Logic: Syntax

Alphabet

(	Left parenthesis	Begin group
)	Right parenthesis	End group
$\neg$	Negation symbol	English: not
$\wedge$	Conjunction symbol	English: and
$\vee$	Disjunction symbol	English: or (inclusive)
$\rightarrow$	Conditional symbol	English: if, then
$\leftrightarrow$	Bi-conditional symbol	English: if and only if
A_1	First propositional symbol	
A_2	Second propositional symbol	
$\ldots$		
A_n	nth propositional symbol	
$\ldots$		

Propositional Logic: Syntax

Alphabet

- *Propositional connective* symbols: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.
- *Logical* symbols: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, **(**, **)**.
- *Parameters* or *nonlogical symbols*: $A_1, A_2, A_3, \ldots$

The meaning of logical symbols is always the same. The meaning of nonlogical symbols depends on the context.

Propositional Logic: Syntax

An *expression* is a sequence of symbols. A sequence is denoted explicitly by a comma separated list enclosed in angle brackets: $<a_1, \ldots, a_m>$.

Examples

$<(, A_1, \wedge, A_3,)>$

$<(, (, \neg, A_1,), \rightarrow, A_2,)>$

$<),), \leftrightarrow,), A_5>$

Propositional Logic: Syntax

An *expression* is a sequence of symbols. A sequence is denoted explicitly by a comma separated list enclosed in angle brackets: $<a_1, \ldots, a_m>$.

Examples

$<(, A_1, \wedge, A_3,)>$	$(A_1 \wedge A_3)$
$<(, (, \neg, A_1,), \rightarrow, A_2,)>$	$((\neg A_1) \rightarrow A_2)$
$<),), \leftrightarrow,), A_5>$	$)) \leftrightarrow) A_5$

For convenience, we will write sequences as a simple string of symbols.

Propositional Logic: Syntax

An *expression* is a sequence of symbols. A sequence is denoted explicitly by a comma separated list enclosed in angle brackets: $<a_1, \ldots, a_m>$.

Examples

$<(, A_1, \wedge, A_3,)>$	$(A_1 \wedge A_3)$
$<(, (, \neg, A_1,), \rightarrow, A_2,)>$	$((\neg A_1) \rightarrow A_2)$
$<),), \leftrightarrow,), A_5>$	$)) \leftrightarrow) A_5$

For convenience, we will write sequences as a simple string of symbols.

Not all expressions make sense. Part of the job of defining a syntax is to *restrict* the kinds of expressions that will be allowed.

Propositional Logic: Well-Formed Formulas

We use a formal inductive definition to define the set W of *well-formed formulas* (wffs) in propositional logic.

- $U =$
- $B =$
- $F =$

Propositional Logic: Well-Formed Formulas

We use a formal inductive definition to define the set W of *well-formed formulas* (wffs) in propositional logic.

- $U =$ the set of all expressions.
- $B =$
- $F =$

Propositional Logic: Well-Formed Formulas

We use a formal inductive definition to define the set W of *well-formed formulas* (wffs) in propositional logic.

- $U =$ the set of all expressions.
- $B =$ the set of expressions consisting of a single propositional symbol.
- $F =$

Propositional Logic: Well-Formed Formulas

We use a formal inductive definition to define the set W of *well-formed formulas* (wffs) in propositional logic.

- $U =$ the set of all expressions.
- $B =$ the set of expressions consisting of a single propositional symbol.
- $F =$ the set of formula-building operations:
  - $\mathcal{E}_{\neg}(\alpha) = (\neg\alpha)$
  - $\mathcal{E}_{\wedge}(\alpha, \beta) = (\alpha \wedge \beta)$
  - $\mathcal{E}_{\vee}(\alpha, \beta) = (\alpha \vee \beta)$
  - $\mathcal{E}_{\rightarrow}(\alpha, \beta) = (\alpha \rightarrow \beta)$
  - $\mathcal{E}_{\leftrightarrow}(\alpha, \beta) = (\alpha \leftrightarrow \beta)$

Propositional Logic: Well-Formed Formulas

We use a formal inductive definition to define the set W of *well-formed formulas* (wffs) in propositional logic.

- $U =$ the set of all expressions.
- $B =$ the set of expressions consisting of a single propositional symbol.
- $F =$ the set of formula-building operations:
  - $\mathcal{E}_{\neg}(\alpha) = (\neg\alpha)$
  - $\mathcal{E}_{\wedge}(\alpha, \beta) = (\alpha \wedge \beta)$
  - $\mathcal{E}_{\vee}(\alpha, \beta) = (\alpha \vee \beta)$
  - $\mathcal{E}_{\rightarrow}(\alpha, \beta) = (\alpha \rightarrow \beta)$
  - $\mathcal{E}_{\leftrightarrow}(\alpha, \beta) = (\alpha \leftrightarrow \beta)$

W is the set generated from F by B.

Propositional Logic: Well-Formed Formulas

We use a formal inductive definition to define the set W of *well-formed formulas* (wffs) in propositional logic.

- $U =$ the set of all expressions.
- $B =$ the set of expressions consisting of a single propositional symbol.
- $F =$ the set of formula-building operations:
  - $\mathcal{E}_{\neg}(\alpha) = (\neg\alpha)$
  - $\mathcal{E}_{\wedge}(\alpha, \beta) = (\alpha \wedge \beta)$
  - $\mathcal{E}_{\vee}(\alpha, \beta) = (\alpha \vee \beta)$
  - $\mathcal{E}_{\rightarrow}(\alpha, \beta) = (\alpha \rightarrow \beta)$
  - $\mathcal{E}_{\leftrightarrow}(\alpha, \beta) = (\alpha \leftrightarrow \beta)$

In fact, W is *freely generated*, meaning there is only one way to generate each member of the set.

Propositional Logic: Semantics

Intuitively, given a *wff* α and a value (either $\mathbf{T}$ or $\mathbf{F}$) for each propositional symbol in α, we should be able to determine the value of α.

Propositional Logic: Semantics

Intuitively, given a *wff* α and a value (either $\mathbf{T}$ or $\mathbf{F}$) for each propositional symbol in α, we should be able to determine the value of α.

How do we make this precise?

Propositional Logic: Semantics

Intuitively, given a *wff* α and a value (either $\mathbf{T}$ or $\mathbf{F}$) for each propositional symbol in α, we should be able to determine the value of α.

How do we make this precise?

Let v be a function from B to $\{\mathbf{F}, \mathbf{T}\}$. We call this function a *truth assignment*.

Propositional Logic: Semantics

Now, we define $\overline{v}$, a function from W to $\{\mathbf{F}, \mathbf{T}\}$ as follows (we compute with $\mathbf{F}$ and $\mathbf{T}$ as if they were 0 and 1 respectively).

- For each propositional symbol A_i, $\overline{v}(A_i) = v(A_i)$.
- $\overline{v}(\mathcal{E}_{\neg}(\alpha)) = \mathbf{T} - \overline{v}(\alpha)$
- $\overline{v}(\mathcal{E}_{\wedge}(\alpha, \beta)) = \min(\overline{v}(\alpha), \overline{v}(\beta))$
- $\overline{v}(\mathcal{E}_{\vee}(\alpha, \beta)) = \max(\overline{v}(\alpha), \overline{v}(\beta))$
- $\overline{v}(\mathcal{E}_{\rightarrow}(\alpha, \beta)) = \max(\mathbf{T} - \overline{v}(\alpha), \overline{v}(\beta))$
- $\overline{v}(\mathcal{E}_{\leftrightarrow}(\alpha, \beta)) = \mathbf{T} - |\overline{v}(\alpha) - \overline{v}(\beta)|$

The fact that W is freely generated ensures that $\overline{v}$ is well-defined.

Truth Tables

There are other ways to present the semantics which are less formal but perhaps more intuitive.

α	$\neg\alpha$
T	
F	

α	β	$\alpha \wedge \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

α	β	$\alpha \vee \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

α	β	$\alpha \rightarrow \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

α	β	$\alpha \leftrightarrow \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

Truth Tables

There are other ways to present the semantics which are less formal but perhaps more intuitive.

α	$\neg\alpha$
T	**F**
F	**T**

α	β	$\alpha \wedge \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

α	β	$\alpha \vee \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

α	β	$\alpha \rightarrow \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

α	β	$\alpha \leftrightarrow \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

Truth Tables

There are other ways to present the semantics which are less formal but perhaps more intuitive.

α	$\neg\alpha$
T	**F**
F	**T**

α	β	$\alpha \wedge \beta$
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**F**

α	β	$\alpha \vee \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

α	β	$\alpha \rightarrow \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

α	β	$\alpha \leftrightarrow \beta$
T	**T**	
T	**F**	
F	**T**	
F	**F**	

Truth Tables

There are other ways to present the semantics which are less formal but perhaps more intuitive.

α	$\neg\alpha$
T	**F**
F	**T**

α	β	$\alpha \land \beta$
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**F**

α	β	$\alpha \lor \beta$
T	**T**	**T**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

α	β	$\alpha \longrightarrow \beta$
T	**T**	**T**
T	**F**	**F**
F	**T**	**T**
F	**F**	**T**

α	β	$\alpha \longleftrightarrow \beta$
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**T**

Complex truth tables

Truth tables can also be used to calculate all possible values of $\overline{v}$ for a given *wff*: We associate a column with each propositional symbol and a column with each propositional connective. There is a row for each possible truth assignment to the propositional connectives.

A_1	A_2	A_3	$(A_1$	$\vee$	$(A_2$	$\wedge$	$\neg A_3))$
T	**T**	**T**	**T**		**T**		
T	**T**	**F**	**T**		**T**		
T	**F**	**T**	**T**		**F**		
T	**F**	**F**	**T**		**F**		
F	**T**	**T**	**F**		**T**		
F	**T**	**F**	**F**		**T**		
F	**F**	**T**	**F**		**F**		
F	**F**	**F**	**F**		**F**		

Complex truth tables

Truth tables can also be used to calculate all possible values of $\overline{v}$ for a given *wff*: We associate a column with each propositional symbol and a column with each propositional connective. There is a row for each possible truth assignment to the propositional connectives.

A_1	A_2	A_3	$(A_1$	$\vee$	$(A_2$	$\wedge$	$\neg A_3))$
T	T	T	T		T		F
T	T	F	T		T		T
T	F	T	T		F		F
T	F	F	T		F		T
F	T	T	F		T		F
F	T	F	F		T		T
F	F	T	F		F		F
F	F	F	F		F		T

Complex truth tables

Truth tables can also be used to calculate all possible values of $\overline{v}$ for a given *wff*: We associate a column with each propositional symbol and a column with each propositional connective. There is a row for each possible truth assignment to the propositional connectives.

A_1	A_2	A_3	$(A_1$	$\vee$	$(A_2$	$\wedge$	$\neg A_3))$
T	**T**	**T**	**T**		**T**	**F**	**F**
T	**T**	**F**	**T**		**T**	**T**	**T**
T	**F**	**T**	**T**		**F**	**F**	**F**
T	**F**	**F**	**T**		**F**	**F**	**T**
F	**T**	**T**	**F**		**T**	**F**	**F**
F	**T**	**F**	**F**		**T**	**T**	**T**
F	**F**	**T**	**F**		**F**	**F**	**F**
F	**F**	**F**	**F**		**F**	**F**	**T**

Complex truth tables

Truth tables can also be used to calculate all possible values of $\overline{v}$ for a given *wff*: We associate a column with each propositional symbol and a column with each propositional connective. There is a row for each possible truth assignment to the propositional connectives.

A_1	A_2	A_3	$(A_1$	$\vee$	$(A_2$	$\wedge$	$\neg A_3))$
T	**T**	**T**	**T**	**T**	**T**	**F**	**F**
T	**T**	**F**	**T**	**T**	**T**	**T**	**T**
T	**F**	**T**	**T**	**T**	**F**	**F**	**F**
T	**F**	**F**	**T**	**T**	**F**	**F**	**T**
F	**T**	**T**	**F**	**F**	**T**	**F**	**F**
F	**T**	**F**	**F**	**T**	**T**	**T**	**T**
F	**F**	**T**	**F**	**F**	**F**	**F**	**F**
F	**F**	**F**	**F**	**F**	**F**	**F**	**T**

Definitions

If α is a *wff*, then a truth assignment v *satisfies* α if $\overline{v}(\alpha) = \mathbf{T}$.

Definitions

If α is a *wff*, then a truth assignment v *satisfies* α if $\overline{v}(\alpha) = \mathbf{T}$.

A *wff* α is *satisfiable* if there exists some truth assignment v which satisfies α.

Definitions

If α is a *wff*, then a truth assignment v *satisfies* α if $\overline{v}(\alpha) = \mathbf{T}$.

A *wff* α is *satisfiable* if there exists some truth assignment v which satisfies α.

Suppose Σ is a set of *wffs*. Then Σ *tautologically implies* α, $\Sigma \models \alpha$, if every truth assignment which satisfies each formula in Σ also satisfies α.

Definitions

If α is a *wff*, then a truth assignment v *satisfies* α if $\overline{v}(\alpha) = \mathbf{T}$.

A *wff* α is *satisfiable* if there exists some truth assignment v which satisfies α.

Suppose Σ is a set of *wffs*. Then Σ *tautologically implies* α, $\Sigma \models \alpha$, if every truth assignment which satisfies each formula in Σ also satisfies α.

- If $\emptyset \models \alpha$, then we say α is a *tautology* or α is *valid* and write $\models \alpha$.

Definitions

If α is a *wff*, then a truth assignment v *satisfies* α if $\overline{v}(\alpha) = \mathbf{T}$.

A *wff* α is *satisfiable* if there exists some truth assignment v which satisfies α.

Suppose Σ is a set of *wffs*. Then Σ *tautologically implies* α, $\Sigma \models \alpha$, if every truth assignment which satisfies each formula in Σ also satisfies α.

- If $\emptyset \models \alpha$, then we say α is a *tautology* or α is *valid* and write $\models \alpha$.
- If Σ is *unsatisfiable*, then $\Sigma \models \alpha$ for every *wff* α.

Definitions

If α is a *wff*, then a truth assignment v *satisfies* α if $\overline{v}(\alpha) = \mathbf{T}$.

A *wff* α is *satisfiable* if there exists some truth assignment v which satisfies α.

Suppose Σ is a set of *wffs*. Then Σ *tautologically implies* α, $\Sigma \models \alpha$, if every truth assignment which satisfies each formula in Σ also satisfies α.

- If $\emptyset \models \alpha$, then we say α is a *tautology* or α is *valid* and write $\models \alpha$.
- If Σ is *unsatisfiable*, then $\Sigma \models \alpha$ for every *wff* α.
- If $\alpha \models \beta$ (shorthand for $\{\alpha\} \models \beta$) and $\beta \models \alpha$, then α and β are *tautologically equivalent*.

Definitions

If α is a *wff*, then a truth assignment v *satisfies* α if $\overline{v}(\alpha) = \mathbf{T}$.

A *wff* α is *satisfiable* if there exists some truth assignment v which satisfies α.

Suppose Σ is a set of *wffs*. Then Σ *tautologically implies* α, $\Sigma \models \alpha$, if every truth assignment which satisfies each formula in Σ also satisfies α.

- If $\emptyset \models \alpha$, then we say α is a *tautology* or α is *valid* and write $\models \alpha$.
- If Σ is *unsatisfiable*, then $\Sigma \models \alpha$ for every *wff* α.
- If $\alpha \models \beta$ (shorthand for $\{\alpha\} \models \beta$) and $\beta \models \alpha$, then α and β are *tautologically equivalent*.
- $\Sigma \models \alpha$ if and only if $\bigwedge(\Sigma) \rightarrow \alpha$ is valid.

Examples

- $(A \lor B) \land (\neg A \lor \neg B)$

Examples

- $(A \lor B) \land (\neg A \lor \neg B)$ is *satisfiable*, but not *valid*.

Examples

- $(A \vee B) \wedge (\neg A \vee \neg B)$ is *satisfiable*, but not *valid*.
- $(A \vee B) \wedge (\neg A \vee \neg B) \wedge (A \leftrightarrow B)$

Examples

- $(A \vee B) \wedge (\neg A \vee \neg B)$ is *satisfiable*, but not *valid*.
- $(A \vee B) \wedge (\neg A \vee \neg B) \wedge (A \leftrightarrow B)$ is *unsatisfiable*.

Examples

- $(A \vee B) \wedge (\neg A \vee \neg B)$ is *satisfiable*, but not *valid*.
- $(A \vee B) \wedge (\neg A \vee \neg B) \wedge (A \leftrightarrow B)$ is *unsatisfiable*.
- $\{A, A \rightarrow B\} \models B$
- $\{A, \neg A\} \models (A \wedge \neg A)$

Examples

- $(A \lor B) \land (\neg A \lor \neg B)$ is *satisfiable*, but not *valid*.
- $(A \lor B) \land (\neg A \lor \neg B) \land (A \leftrightarrow B)$ is *unsatisfiable*.
- $\{A, A \rightarrow B\} \models B$
- $\{A, \neg A\} \models (A \land \neg A)$
- $\neg(A \land B)$ is *tautologically equivalent* to $\neg A \lor \neg B$

Examples

- $(A \vee B) \wedge (\neg A \vee \neg B)$ is *satisfiable*, but not *valid*.
- $(A \vee B) \wedge (\neg A \vee \neg B) \wedge (A \leftrightarrow B)$ is *unsatisfiable*.
- $\{A, A \rightarrow B\} \models B$
- $\{A, \neg A\} \models (A \wedge \neg A)$
- $\neg(A \wedge B)$ is *tautologically equivalent* to $\neg A \vee \neg B$

Suppose you had an algorithm *SAT* which would take a *wff* α as input and return *True* if α is satisfiable and *False* otherwise.

How would you use this algorithm to verify each of the claims made above?

Examples

- $(A \vee B) \wedge (\neg A \vee \neg B)$ is *satisfiable*, but not *valid*.
- $(A \vee B) \wedge (\neg A \vee \neg B) \wedge (A \leftrightarrow B)$ is *unsatisfiable*.
- $\{A, A \rightarrow B\} \models B$ $\qquad (A \wedge (A \rightarrow B) \wedge (\neg B))$
- $\{A, \neg A\} \models (A \wedge \neg A)$
- $\neg(A \wedge B)$ is *tautologically equivalent* to $\neg A \vee \neg B$

Suppose you had an algorithm *SAT* which would take a *wff* α as input and return *True* if α is satisfiable and *False* otherwise.

How would you use this algorithm to verify each of the claims made above?

Examples

- $(A \vee B) \wedge (\neg A \vee \neg B)$ is *satisfiable*, but not *valid*.
- $(A \vee B) \wedge (\neg A \vee \neg B) \wedge (A \leftrightarrow B)$ is *unsatisfiable*.
- $\{A, A \rightarrow B\} \models B$ $\qquad (A \wedge (A \rightarrow B) \wedge (\neg B))$
- $\{A, \neg A\} \models (A \wedge \neg A)$ $\qquad (A \wedge (\neg A) \wedge \neg(A \wedge \neg A))$
- $\neg(A \wedge B)$ is *tautologically equivalent* to $\neg A \vee \neg B$

Suppose you had an algorithm *SAT* which would take a *wff* α as input and return *True* if α is satisfiable and *False* otherwise.

How would you use this algorithm to verify each of the claims made above?

Examples

- $(A \vee B) \wedge (\neg A \vee \neg B)$ is *satisfiable*, but not *valid*.
- $(A \vee B) \wedge (\neg A \vee \neg B) \wedge (A \leftrightarrow B)$ is *unsatisfiable*.
- $\{A, A \rightarrow B\} \models B$ $\qquad (A \wedge (A \rightarrow B) \wedge (\neg B))$
- $\{A, \neg A\} \models (A \wedge \neg A)$ $\qquad (A \wedge (\neg A) \wedge \neg(A \wedge \neg A))$
- $\neg(A \wedge B)$ is *tautologically equivalent* to $\neg A \vee \neg B$
  $\neg(\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B))$

Suppose you had an algorithm *SAT* which would take a *wff* α as input and return *True* if α is satisfiable and *False* otherwise.

How would you use this algorithm to verify each of the claims made above?

Some tautologies

Associative and Commutative laws for $\wedge, \vee, \leftrightarrow$

Distributive Laws

- $(A \wedge (B \vee C)) \leftrightarrow ((A \wedge B) \vee (A \wedge C))$.
- $(A \vee (B \wedge C)) \leftrightarrow ((A \vee B) \wedge (A \vee C))$.

De Morgan's Laws

- $\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$
- $\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$

Implication

- $(A \rightarrow B) \leftrightarrow (\neg A \vee B)$

Determining Satisfiability using Truth Tables

An Algorithm for Satisfiability

To check whether α is satisfiable, form the truth table for α. If there is a row in which $\mathbf{T}$ appears as the value for α, then α is *satisfiable*. Otherwise, α is *unsatisfiable*.

Determining Satisfiability using Truth Tables

An Algorithm for Satisfiability

To check whether α is satisfiable, form the truth table for α. If there is a row in which $\mathbf{T}$ appears as the value for α, then α is *satisfiable*. Otherwise, α is *unsatisfiable*.

An Algorithm for Tautological Implication

To check whether $\{\alpha_1, \ldots, \alpha_k\} \models \beta$, check the satisfiability of $(\alpha_1 \wedge \cdots \wedge \alpha_k) \wedge (\neg\beta)$. If it is *unsatisfiable*, then $\{\alpha_1, \ldots, \alpha_k\} \models \beta$, otherwise $\{\alpha_1, \ldots, \alpha_k\} \not\models \beta$.

Determining Satisfiability using Truth Tables

Example

$$A \wedge ((B \vee \neg A) \wedge (C \vee \neg B))$$

Determining Satisfiability using Truth Tables

Example

$A \wedge ((B \vee \neg A) \wedge (C \vee \neg B))$

A	B	C	A	$\wedge$	$((B$	$\vee$	$\neg A)$	$\wedge$	$(C$	$\vee$	$\neg B))$

Determining Satisfiability using Truth Tables

Example

$$A \wedge ((B \vee \neg A) \wedge (C \vee \neg B))$$

A	B	C	A	$\wedge$	$((B$	$\vee$	$\neg A)$	$\wedge$	$(C$	$\vee$	$\neg B))$
F	**F**	**F**		**F**		**T**	**T**	**T**		**T**	**T**

Determining Satisfiability using Truth Tables

Example

$$A \land ((B \lor \neg A) \land (C \lor \neg B))$$

A	B	C	A	$\land$	$((B$	$\lor$	$\neg A)$	$\land$	$(C$	$\lor$	$\neg B))$
F	**F**	**F**		**F**		**T**	**T**	**T**		**T**	**T**
F	**F**	**T**		**F**		**T**	**T**	**T**		**T**	**T**

Determining Satisfiability using Truth Tables

Example

$$A \wedge ((B \vee \neg A) \wedge (C \vee \neg B))$$

A	B	C	A	$\wedge$	$((B$	$\vee$	$\neg A)$	$\wedge$	$(C$	$\vee$	$\neg B))$
F	**F**	**F**		**F**		**T**	**T**	**T**		**T**	**T**
F	**F**	**T**		**F**		**T**	**T**	**T**		**T**	**T**
F	**T**	**F**		**F**		**T**	**T**	**F**		**F**	**F**

Determining Satisfiability using Truth Tables

Example

$$A \land ((B \lor \neg A) \land (C \lor \neg B))$$

A	B	C	A	$\land$	$((B$	$\lor$	$\neg A)$	$\land$	$(C$	$\lor$	$\neg B))$
F	**F**	**F**		**F**		**T**	**T**	**T**		**T**	**T**
F	**F**	**T**		**F**		**T**	**T**	**T**		**T**	**T**
F	**T**	**F**		**F**		**T**	**T**	**F**		**F**	**F**
F	**T**	**T**		**F**		**T**	**T**	**T**		**T**	**F**

Determining Satisfiability using Truth Tables

Example

$A \wedge ((B \vee \neg A) \wedge (C \vee \neg B))$

A	B	C	A	$\wedge$	$((B$	$\vee$	$\neg A)$	$\wedge$	$(C$	$\vee$	$\neg B))$
F	**F**	**F**		**F**		**T**	**T**	**T**		**T**	**T**
F	**F**	**T**		**F**		**T**	**T**	**T**		**T**	**T**
F	**T**	**F**		**F**		**T**	**T**	**F**		**F**	**F**
F	**T**	**T**		**F**		**T**	**T**	**T**		**T**	**F**
T	**F**	**F**		**F**		**F**	**F**	**F**		**T**	**T**

Determining Satisfiability using Truth Tables

Example

$$A \wedge ((B \vee \neg A) \wedge (C \vee \neg B))$$

A	B	C	A	$\wedge$	$((B$	$\vee$	$\neg A)$	$\wedge$	$(C$	$\vee$	$\neg B))$
F	**F**	**F**		**F**		**T**	**T**	**T**		**T**	**T**
F	**F**	**T**		**F**		**T**	**T**	**T**		**T**	**T**
F	**T**	**F**		**F**		**T**	**T**	**F**		**F**	**F**
F	**T**	**T**		**F**		**T**	**T**	**T**		**T**	**F**
T	**F**	**F**		**F**		**F**	**F**	**F**		**T**	**T**
T	**F**	**T**		**F**		**F**	**F**	**F**		**T**	**T**

Determining Satisfiability using Truth Tables

Example

$$A \wedge ((B \vee \neg A) \wedge (C \vee \neg B))$$

A	B	C	A	$\wedge$	$((B$	$\vee$	$\neg A)$	$\wedge$	$(C$	$\vee$	$\neg B))$
F	**F**	**F**		**F**		**T**	**T**	**T**		**T**	**T**
F	**F**	**T**		**F**		**T**	**T**	**T**		**T**	**T**
F	**T**	**F**		**F**		**T**	**T**	**F**		**F**	**F**
F	**T**	**T**		**F**		**T**	**T**	**T**		**T**	**F**
T	**F**	**F**		**F**		**F**	**F**	**F**		**T**	**T**
T	**F**	**T**		**F**		**F**	**F**	**F**		**T**	**T**
T	**T**	**F**		**F**		**T**	**F**	**F**		**F**	**F**

Determining Satisfiability using Truth Tables

Example

$$A \wedge ((B \vee \neg A) \wedge (C \vee \neg B))$$

A	B	C	A	$\wedge$	$((B$	$\vee$	$\neg A)$	$\wedge$	$(C$	$\vee$	$\neg B))$
F	**F**	**F**		**F**		**T**	**T**	**T**		**T**	**T**
F	**F**	**T**		**F**		**T**	**T**	**T**		**T**	**T**
F	**T**	**F**		**F**		**T**	**T**	**F**		**F**	**F**
F	**T**	**T**		**F**		**T**	**T**	**T**		**T**	**F**
T	**F**	**F**		**F**		**F**	**F**	**F**		**T**	**T**
T	**F**	**T**		**F**		**F**	**F**	**F**		**T**	**T**
T	**T**	**F**		**F**		**T**	**F**	**F**		**F**	**F**
T	**T**	**T**		**T**		**T**	**F**	**T**		**T**	**F**

Determining Satisfiability using Truth Tables

What is the complexity of this algorithm?

Determining Satisfiability using Truth Tables

What is the complexity of this algorithm?

2^n where n is the number of propositional symbols.

Determining Satisfiability using Truth Tables

What is the complexity of this algorithm?

- 2^n where n is the number of propositional symbols.

Can we do better?

Determining Satisfiability using Truth Tables

What is the complexity of this algorithm?

2^n where n is the number of propositional symbols.

Can we do better?

SAT was the first problem shown to be $\mathcal{NP}$-*complete* [Coo71]: all of the problems in the class $\mathcal{NP}$ can be solved by translating them (in polynomial time) into SAT.

Determining Satisfiability using Truth Tables

What is the complexity of this algorithm?

2^n where n is the number of propositional symbols.

Can we do better?

SAT was the first problem shown to be $\mathcal{NP}$*-complete* [Coo71]: all of the problems in the class $\mathcal{NP}$ can be solved by translating them (in polynomial time) into SAT.

So, if we could somehow build a *fast* solver for SAT, it could be used to solve lots of other problems.

Determining Satisfiability using Truth Tables

What is the complexity of this algorithm?

2^n where n is the number of propositional symbols.

Can we do better?

SAT was the first problem shown to be $\mathcal{NP}$-*complete* [Coo71]: all of the problems in the class $\mathcal{NP}$ can be solved by translating them (in polynomial time) into SAT.

So, if we could somehow build a *fast* solver for SAT, it could be used to solve lots of other problems.

In theory, this seems dubious, as problems in $\mathcal{NP}$ are known to take exponential time in the worst case.

Determining Satisfiability using Truth Tables

What is the complexity of this algorithm?

2^n where n is the number of propositional symbols.

Can we do better?

SAT was the first problem shown to be $\mathcal{NP}$*-complete* [Coo71]: all of the problems in the class $\mathcal{NP}$ can be solved by translating them (in polynomial time) into SAT.

So, if we could somehow build a *fast* solver for SAT, it could be used to solve lots of other problems.

In theory, this seems dubious, as problems in $\mathcal{NP}$ are known to take exponential time in the worst case.

Remarkably, modern SAT solvers are very fast most of the time!

Roadmap

Boolean Satisfiability

- Propositional Logic
- Solving SAT
- Modeling for SAT

Converting to CNF

Given an arbitrary formula in propostitional logic, most algorithms for determining satisfiability first convert the formula into *conjunctive normal form (CNF)*.

Some definitions:

- A *literal* is a propositional variable or its negation
- A *clause* is a disjunction of one or more literals
- A formula is in *CNF* if it consists of a conjunction of clauses
- A propositional symbol occurs *positively* if it occurs unnegated in a clause.
- A propositional symbol occurs *negatively* if it occurs negated in a clause.

Converting to CNF

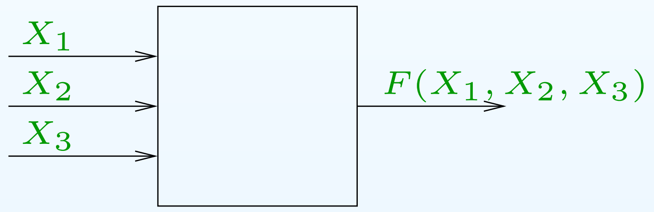
Examples

- Literals: P_i, $\neg P_i$
- Clauses: $(P_1 \vee \neg P_3 \vee P_5)$, $(P_2 \vee \neg P_2)$
- CNF: $(P_1 \vee \neg P_3) \wedge (\neg P_2 \vee P_3 \vee P_5)$
- In the above formula, P_1 occurs positively and P_2 occurs negatively

To provide intuition for how to convert to CNF, we first explore the connection between propositional formulas and Boolean circuits.

Boolean Gates

Consider an electrical device having n inputs and one output. Assume that to each input we apply a signal that is either $\mathbf{T}$ or $\mathbf{F}$, and that this uniquely determines whether the output is $\mathbf{T}$ or $\mathbf{F}$.



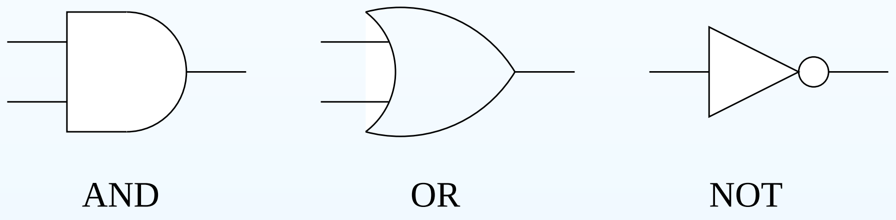
The behavior of such a device is described by a Boolean function:

$F(X_1, \ldots, X_n) =$ the output signal given the input signals $X_1, \ldots, X_n$.

We call such a device a *Boolean gate*.

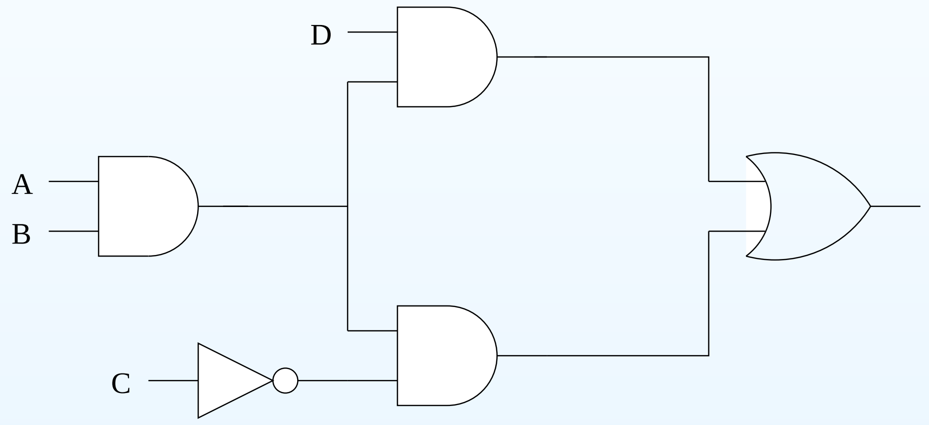
Boolean Gates

Some common Boolean gates include *AND*, *OR*, and *NOT* gates.



Boolean Circuits

The inputs and outputs of Boolean gates can be connected together to form a *combinational Boolean circuit*.

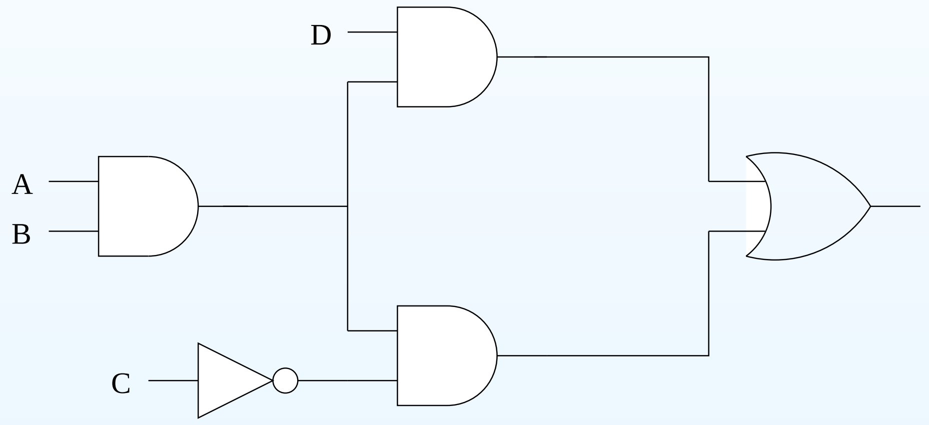


A combinational Boolean circuit corresponds to a *directed acyclic graph* (DAG) whose leaves are *inputs* and each of whose nodes is labeled with the name of a Boolean gate.

One or more of the nodes may be identified as *outputs*.

Boolean Circuits

The inputs and outputs of Boolean gates can be connected together to form a *combinational Boolean circuit*.



There is a natrual correspondence between Boolean circuits and formulas of propositional logic. The formula corresponding to the above circuit is:

$(D \wedge (A \wedge B)) \vee ((A \wedge B) \wedge \neg C)$.

Sharing Sub-Expressions

$$(D \wedge (A \wedge B)) \vee ((A \wedge B) \wedge \neg C)$$

This formula highlights an inefficiency in the logic representation as compared with the circuit representation.

Sharing Sub-Expressions

$(D \wedge (A \wedge B)) \vee ((A \wedge B) \wedge \neg C)$

This formula highlights an inefficiency in the logic representation as compared with the circuit representation.

If we are only concerned with the *satisfiability* of the formula, we can overcome this inefficiency by introducing new propositional symbols:

$((D \wedge E) \vee (E \wedge \neg C)) \wedge (E \leftrightarrow (A \wedge B))$

Sharing Sub-Expressions

$(D \land (A \land B)) \lor ((A \land B) \land \neg C)$

This formula highlights an inefficiency in the logic representation as compared with the circuit representation.

If we are only concerned with the *satisfiability* of the formula, we can overcome this inefficiency by introducing new propositional symbols:

$((D \land E) \lor (E \land \neg C)) \land (E \leftrightarrow (A \land B))$

Note that the new formula is *not* tautologically equivalent to the original formula: *why?*

Sharing Sub-Expressions

$(D \wedge (A \wedge B)) \vee ((A \wedge B) \wedge \neg C)$

This formula highlights an inefficiency in the logic representation as compared with the circuit representation.

If we are only concerned with the *satisfiability* of the formula, we can overcome this inefficiency by introducing new propositional symbols:

$((D \wedge E) \vee (E \wedge \neg C)) \wedge (E \leftrightarrow (A \wedge B))$

Note that the new formula is *not* tautologically equivalent to the original formula: *why?*

But it *is equisatisfiable*: the original formula is satisfiable iff the new formula is satisfiable.

Converting to CNF

This same idea is behind a simple algorithm for converting any formula to CNF [Tse70].

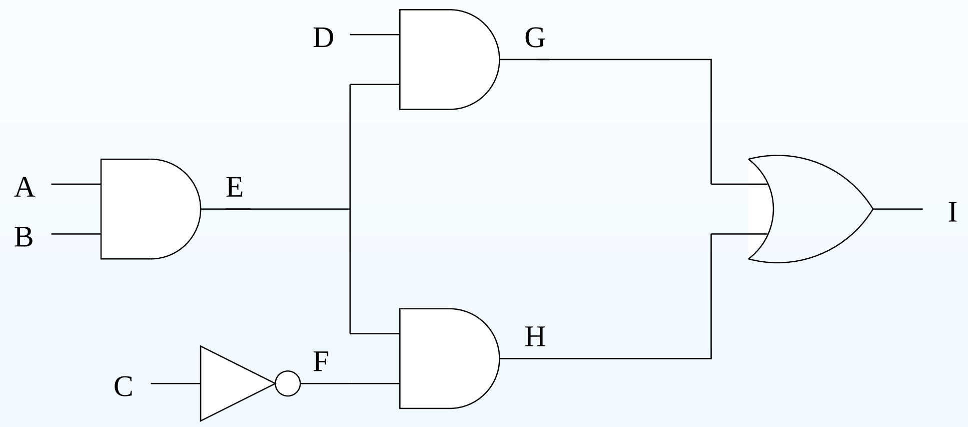
We view the formula as a directed acyclic graph (DAG).

Conversion to CNF

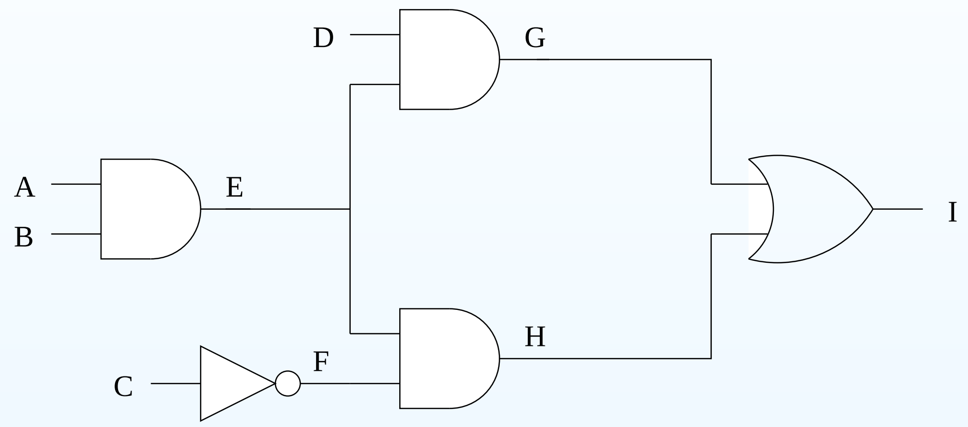
1. Label each non-leaf node of the DAG with a new propositional symbol.
2. For each non-leaf node, construct a conjunction of clauses relating the inputs of that node to its output.
3. Take the conjunction of all of these clauses together with a single clause consisting of the symbol for the root node.

The resulting formula is satisfiable iff the original formula is satisfiable.

Converting to CNF: Example

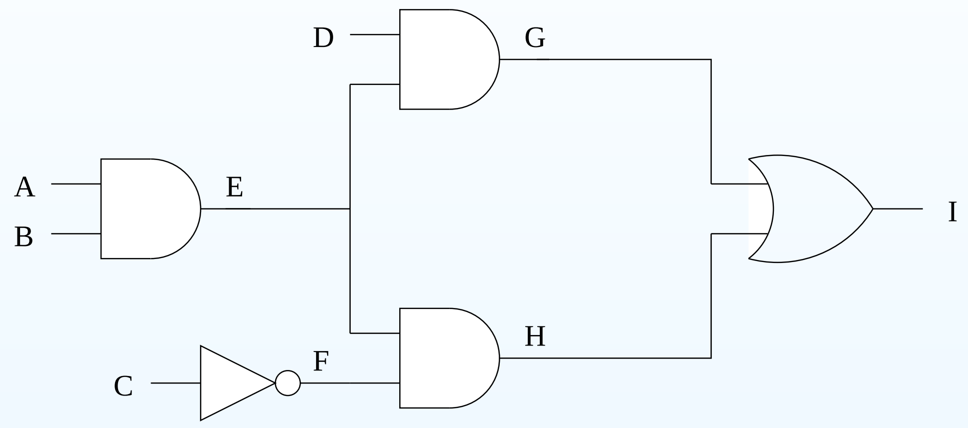


Converting to CNF: Example



$$(A \wedge B) \leftrightarrow E$$

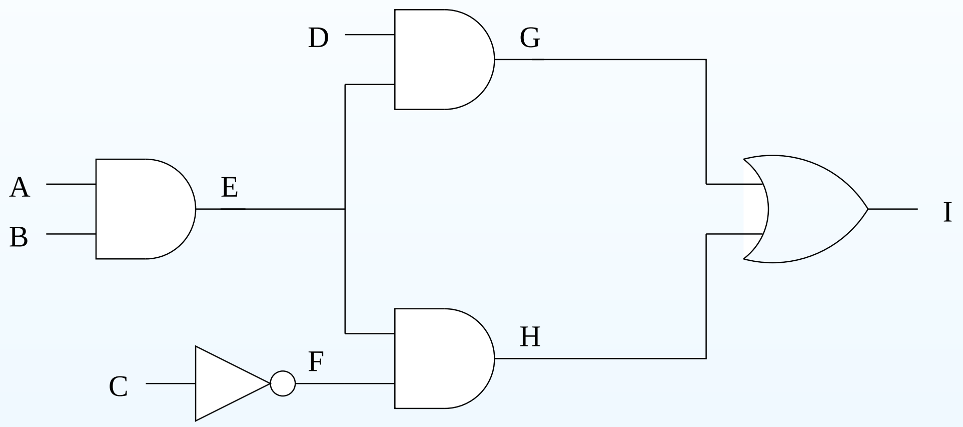
Converting to CNF: Example



$(A \wedge B) \leftrightarrow E$

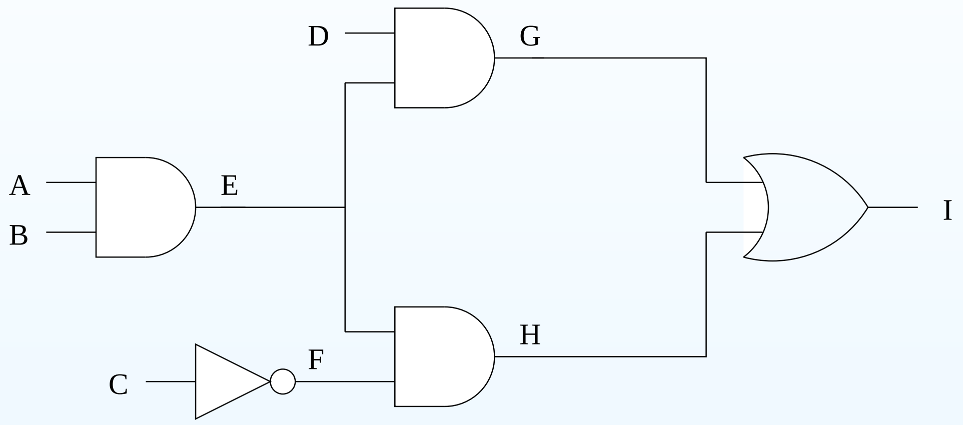
$((A \wedge B) \rightarrow E) \wedge (E \rightarrow (A \wedge B))$

Converting to CNF: Example



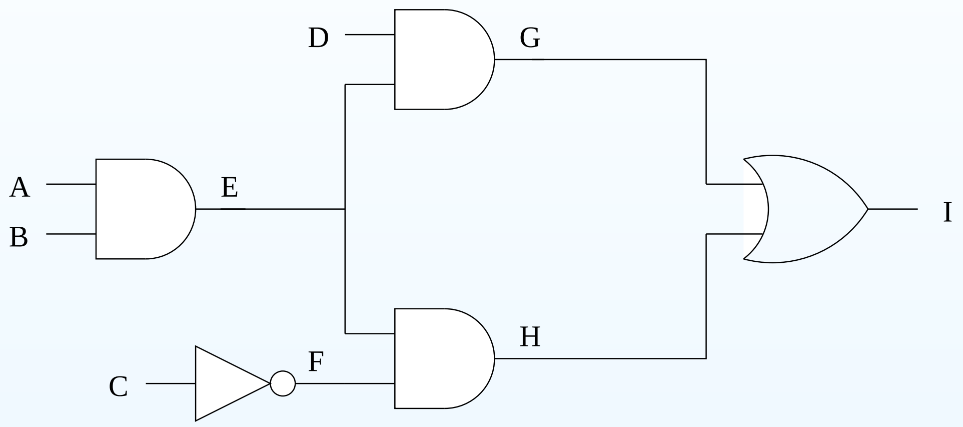
$$(A \wedge B) \leftrightarrow E$$
$$((A \wedge B) \rightarrow E) \wedge (E \rightarrow (A \wedge B))$$
$$(\neg(A \wedge B) \vee E) \wedge (\neg E \vee (A \wedge B))$$

Converting to CNF: Example



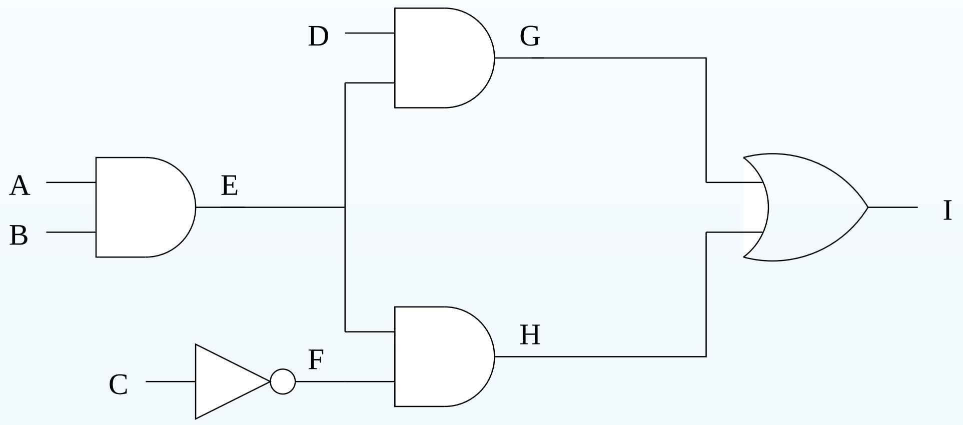
$$(A \wedge B) \leftrightarrow E$$
$$((A \wedge B) \rightarrow E) \wedge (E \rightarrow (A \wedge B))$$
$$(\neg(A \wedge B) \vee E) \wedge (\neg E \vee (A \wedge B))$$
$$(\neg A \vee \neg B \vee E) \wedge (\neg E \vee A) \wedge (\neg E \vee B)$$

Converting to CNF: Example



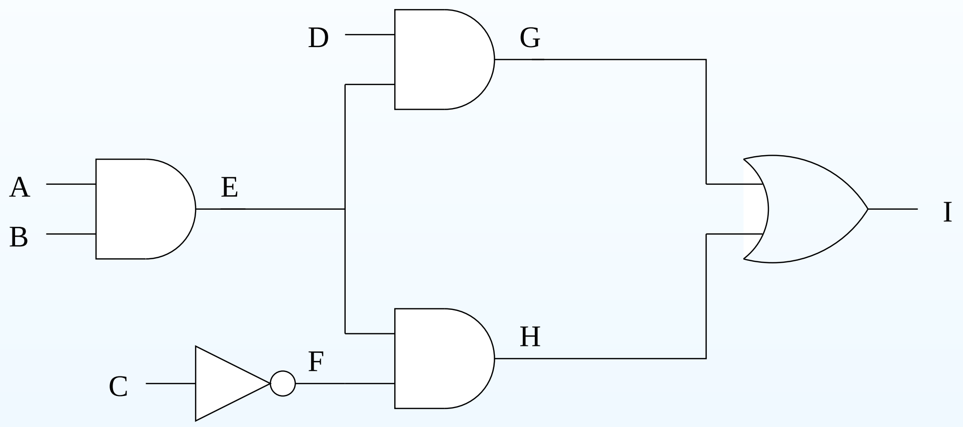
$$(\neg A \vee \neg B \vee E) \wedge (\neg E \vee A) \wedge (\neg E \vee B) \wedge$$

Converting to CNF: Example



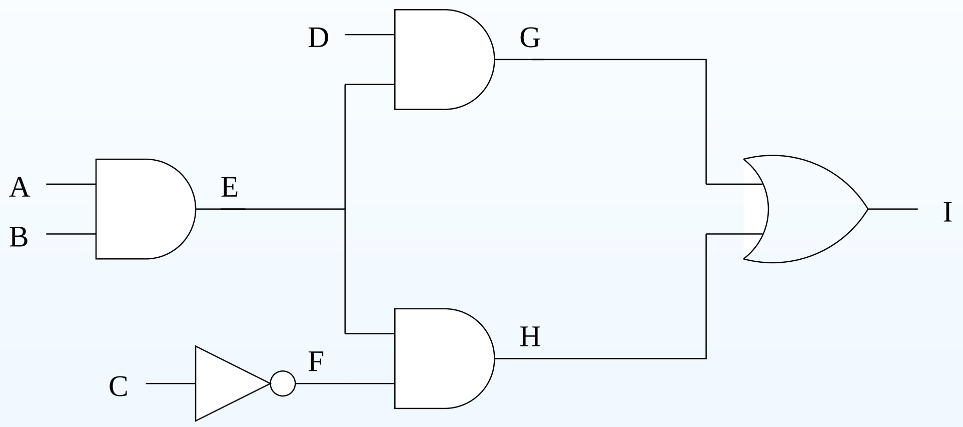
$$(\neg A \vee \neg B \vee E) \wedge (\neg E \vee A) \wedge (\neg E \vee B) \wedge$$
$$(\neg C \vee F) \wedge (\neg F \vee C) \wedge$$

Converting to CNF: Example



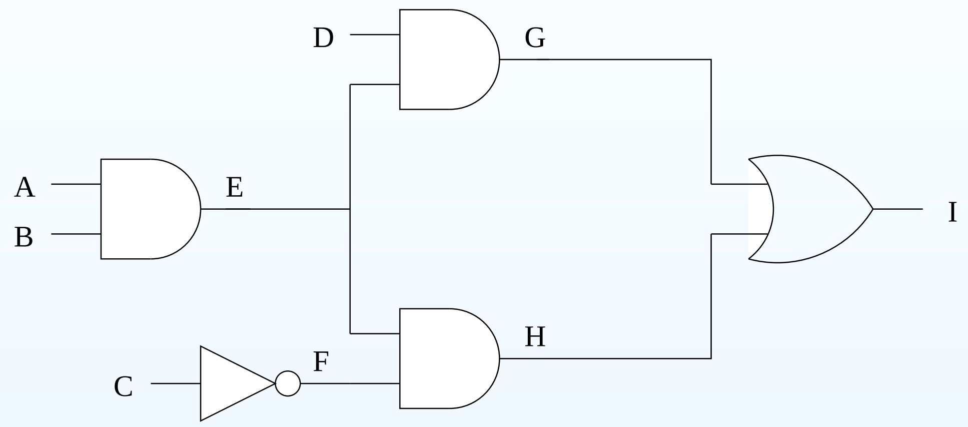
$$(\neg A \vee \neg B \vee E) \wedge (\neg E \vee A) \wedge (\neg E \vee B) \wedge$$
$$(\neg C \vee F) \wedge (\neg F \vee C) \wedge$$
$$(\neg D \vee \neg E \vee G) \wedge (\neg G \vee D) \wedge (\neg G \vee E) \wedge$$

Converting to CNF: Example



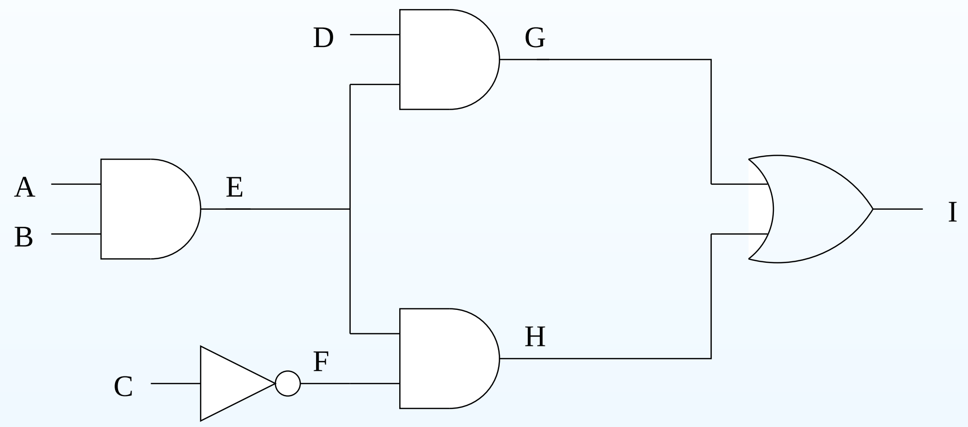
$$(\neg A \lor \neg B \lor E) \land (\neg E \lor A) \land (\neg E \lor B) \land$$
$$(\neg C \lor F) \land (\neg F \lor C) \land$$
$$(\neg D \lor \neg E \lor G) \land (\neg G \lor D) \land (\neg G \lor E) \land$$
$$(\neg E \lor \neg F \lor H) \land (\neg H \lor E) \land (\neg H \lor F) \land$$

Converting to CNF: Example



$$(\neg A \vee \neg B \vee E) \wedge (\neg E \vee A) \wedge (\neg E \vee B) \wedge$$
$$(\neg C \vee F) \wedge (\neg F \vee C) \wedge$$
$$(\neg D \vee \neg E \vee G) \wedge (\neg G \vee D) \wedge (\neg G \vee E) \wedge$$
$$(\neg E \vee \neg F \vee H) \wedge (\neg H \vee E) \wedge (\neg H \vee F) \wedge$$
$$(G \vee H \vee \neg I) \wedge (I \vee \neg G) \wedge (I \vee \neg H) \wedge$$

Converting to CNF: Example



$$
\begin{aligned}
&(\neg A \vee \neg B \vee E) \wedge (\neg E \vee A) \wedge (\neg E \vee B) \wedge \\
&(\neg C \vee F) \wedge (\neg F \vee C) \wedge \\
&(\neg D \vee \neg E \vee G) \wedge (\neg G \vee D) \wedge (\neg G \vee E) \wedge \\
&(\neg E \vee \neg F \vee H) \wedge (\neg H \vee E) \wedge (\neg H \vee F) \wedge \\
&(G \vee H \vee \neg I) \wedge (I \vee \neg G) \wedge (I \vee \neg H) \wedge \\
&(I)
\end{aligned}
$$

CNF: Alternative notations

$$(\neg A \lor \neg B \lor E) \land (\neg E \lor A) \land (\neg E \lor B) \land$$
$$(\neg C \lor F) \land (\neg F \lor C) \land$$
$$(\neg D \lor \neg E \lor G) \land (\neg G \lor D) \land (\neg G \lor E) \land$$
$$(\neg E \lor \neg F \lor H) \land (\neg H \lor E) \land (\neg H \lor F) \land$$
$$(G \lor H \lor \neg I) \land (I \lor \neg G) \land (I \lor \neg H) \land$$
$$(I)$$

$$(A' + B' + E)(E' + A)(E' + B)$$
$$(C' + F)(F' + C)$$
$$(D' + E' + G)(G' + D)(G' + E)$$
$$(E' + F' + H)(H' + E)(H' + F)$$
$$(G + H + I')(I + G')(I + H')$$
$$(I)$$

CNF: Alternative notations

DIMACS standard

Each variable is represented by a positive integer. A negative integer refers to the negation of the variable. Clauses are given as sequences of integers separated by spaces. A 0 terminates the clause.

$(A' + B' + E)(E' + A)(E' + B)$	-1 -2 5 0	-5 1 0	-5 2 0
$(C' + F)(F' + C)$	-3 6 0	-6 3 0	
$(D' + E' + G)(G' + D)(G' + E)$	-4 -5 7 0	-7 4 0	-7 5 0
$(E' + F' + H)(H' + E)(H' + F)$	-5 -6 8 0	-8 5 0	-8 6 0
$(G + H + I')(I + G')(I + H')$	7 8 -9 0	9 -7 0	9 -8 0
(I)	9 0		

Davis-Putnam Algorithm

From now on, unless otherwise indicated, we assume formulas are in CNF, or, equivalently, that we have a set of clauses to check for satisfiability (i.e. the conjunction is implicit).

The first algorithm to try something more sophisticated than the truth-table method was the *Davis-Putnam (DP)* algorithm, published in 1960 [DP60].

It is often confused with the later, more popular algorithm presented by Davis, Logemann, and Loveland in 1962 [DLL62], which we will refer to as *Davis-Putnam-Logemann-Loveland (DPLL)*.

We first consider the original DP algorithm.

Davis-Putnam Algorithm

There are three satisfiability-preserving transformations in DP.

- The 1-literal rule
- The affirmative-negative rule
- The rule for eliminating atomic formulas

The first two steps reduce the total number of literals in the formula.

The last step reduces the number of variables in the formula.

By repeatedly applying these rules, eventually we obtain a formula containing an empty clause, indicating unsatisfiability, or a formula with no clauses, indicating satisfiability.

Davis-Putnam Algorithm

The 1-literal rule

Also called *unit propagation*.

Suppose (p) is a unit clause (clause containing only one literal). Let $-p$ denote the negation of p where double negation is collapsed (i.e. $-\neg q \equiv q$).

- Remove all instances of $-p$ from clauses in the formula (shortening the corresponding clauses).
- Remove all clauses containing p (including the unit clause itself).

Davis-Putnam Algorithm

The affirmative-negative rule

Also called the *pure literal rule*.

If a literal appears *only positively* or *only negatively*, delete all clauses containing that literal.

Why does this preserve satisfiability?

Davis-Putnam Algorithm

Rule for eliminating atomic formulas

Also called the *resolution rule*.

- Choose a propositional symbol p which occurs positively in at least one clause and negatively in at least one other clause.
- Let P be the set of all clauses in which p occurs positively.
- Let N be the set of all clauses in which p occurs negatively.
- Replace the clauses in P and N with those obtained by resolution on p using all pairs of clauses from P and N.

For a single pair of clauses, $(p \vee l_1 \vee \cdots \vee l_m)$ and $(\neg p \vee k_1 \vee \cdots \vee k_n)$, *resolution on* p forms the new clause $(l_1 \vee \cdots \vee l_m \vee k_1 \vee \cdots \vee k_n)$.

DPLL Algorithm

In the worst case, the resolution rule can cause a quadratic expansion every time it is applied.

For large formulas, this can quickly exhaust the available memory.

The DPLL algorithm replaces resolution with a *splitting rule*.

- Choose a propositional symbol p occuring in the formula.
- Let Δ be the current set of clauses.
- Test the satisfiability of $\Delta \cup \{(p)\}$.
- If satisfiable, return *True*.
- Otherwise, return the result of testing $\Delta \cup \{(\neg p)\}$ for satisfiability.

Some Experimental Results [Har09]

Problem	tautology	dptaut	dplltaut
prime 3	0.00	0.00	0.00
prime 4	0.02	0.06	0.04
prime 9	18.94	2.98	0.51
prime 10	11.40	3.03	0.96
prime 11	28.11	2.98	0.51
prime 16	$>$1 hour	out of memory	9.15
prime 17	$>$1 hour	out of memory	3.87
ramsey 3 3 5	0.03	0.06	0.02
ramsey 3 3 6	5.13	8.28	0.31
mk_adder_test 3 2	$>>$1 hour	6.50	7.34
mk_adder_test 4 2	$>>$1 hour	22.95	46.86
mk_adder_test 5 2	$>>$1 hour	44.83	170.98
mk_adder_test 5 3	$>>$1 hour	38.27	250.16
mk_adder_test 6 3	$>>$1 hour	out of memory	1186.4
mk_adder_test 7 3	$>>$1 hour	out of memory	3759.9

DPLL Algorithm

The DPLL algorithm is the basis for most modern SAT solvers.

We will look at DPLL in more detail, but first we consider two more alternative algorithms.

Incomplete SAT: GSAT [SLM92]

```
Input: a set of clauses $F$, MAX-FLIPS, MAX-TRIES
Output: a satisfying truth assignment of $F$
        or $\emptyset$, if none found
for $i$ := 1 to MAX-TRIES
  $v$ := a randomly generated truth assignment
  for $j$ := 1 to MAX-FLIPS
    if $v$ satisfies $F$ then return $v$
    $p$ := a propositional variable such that a
           change in its truth assignment gives the
           largest increase in the total number of
           clauses of $F$ that are satisfied by $v$
    $v$ := $v$ with the assignment to $p$ reversed
  end for
end for
return $\emptyset$
```

Stålmarck's Method [SS98]

Breadth-first approach instead of depth-first.

Dilemma Rule

Given a set of formulas Δ and any basic deduction algorithm, R, the dilemma rule performs a case split on some literal p by considering the new sets of formulas $\Delta \cup \{(\neg p)\}$ and $\Delta \cup \{(p)\}$.

To each of these sets, the algorithm R is applied to yield Δ_0 and Δ_1 respectively.

The original set Δ is then augmented with $\Delta_0 \cap \Delta_1$.

Stålmarck's Method [SS98]

Breadth-first approach instead of depth-first.

Dilemma Rule

Given a set of formulas Δ and any basic deduction algorithm, R, the dilemma rule performs a case split on some literal p by considering the new sets of formulas $\Delta \cup \{(\neg p)\}$ and $\Delta \cup \{(p)\}$.

To each of these sets, the algorithm R is applied to yield Δ_0 and Δ_1 respectively.

The original set Δ is then augmented with $\Delta_0 \cap \Delta_1$.

In 1994, Kunz and Pradhan developed a technique they called *recursive learning* which is very similar to the dilemma rule [KP94].

Stålmarck's Method

Stålmarck's Method takes as input a set of formulas Δ and a set of basic deduction rules S_0.

Applying S_0 to Δ until no further deductions are possible is called *0-saturation*.

Applying the dilemma rule with $R = S_0$ until no further deductions are possible is called *1-saturation*, and the result is denoted S_1. Note that in order to acheive 1-saturation, the dilemma rule is applied for *every* variable. This is why Stålmarck's Method can be classified as a breadth-first strategy.

Repeatedly applying the dilemma rule with $R = S_1$ is called *2-saturation*, and denoted S_2.

In general, S_{n+1} or $(n+1)$*-saturation* is obtained by applying the dilemma rule with $R = S_n$.

Stålmarck's Method

If a set of formulas Δ is decidable by n-saturation, then Δ is said to be n-*easy*. If, in addition, it is not decidable by $(n-1)$-saturation, it is said to be n-*hard*.

If Δ contains at most n propositional symbols, then Δ is clearly n-easy. *Why?*

Stålmarck's Method

If a set of formulas Δ is decidable by n-saturation, then Δ is said to be n*-easy*. If, in addition, it is not decidable by $(n-1)$-saturation, it is said to be n*-hard*.

If Δ contains at most n propositional symbols, then Δ is clearly n-easy. *Why?*

The merit of Stålmarck's method is that for some applications, the problems are nearly always n-easy for small values of n, often just $n = 1$.

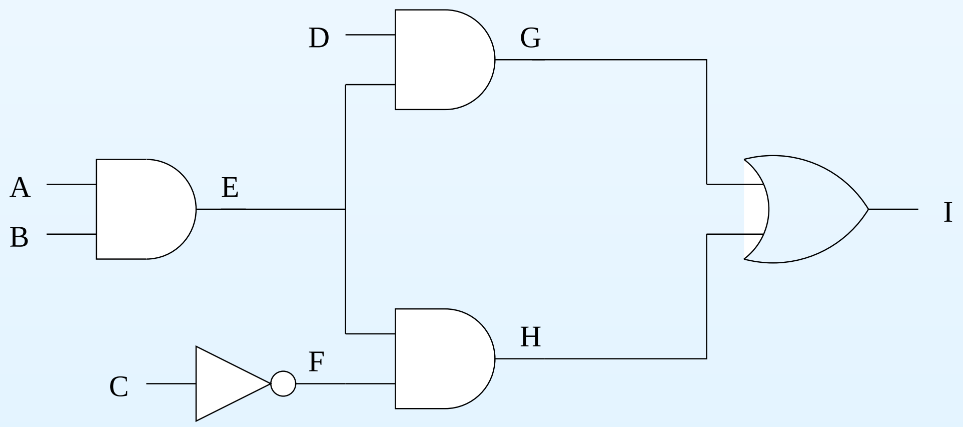
Stålmarck's Method: Implementation

Triplets

Stålmarck's Method does not use CNF. Instead, it first translates a formula into a set of *triplets*: $p_i \leftrightarrow p_j \bowtie p_k$.

The translation is analagous to the conversion to CNF except that the equivalences for each node are not transformed into clauses: they are left as equivalences.

Example



$(E \leftrightarrow A \wedge B)$,$(G \leftrightarrow D \wedge E)$, $(H \leftrightarrow E \wedge \neg C)$, $(I \leftrightarrow G \vee H)$

Stålmarck’s Method: Implementation

Simple Rules

The rules for 0-saturation simply enumerate the new equivalences that can be deduced from a triplet given a set of existing equivalences.

Example

Consider the triplet $p \leftrightarrow q \wedge r$

- If $r \leftrightarrow$ *True*, then $p \leftrightarrow q$.
- If $p \leftrightarrow$ *True*, then $q \leftrightarrow$ *True* and $r \leftrightarrow$ *True*.
- If $q \leftrightarrow$ *False*, then $p \leftrightarrow$ *False*.
- If $q \leftrightarrow r$, then $p \leftrightarrow q$ and $p \leftrightarrow r$.
- If $p \leftrightarrow \neg q$, then $q \leftrightarrow$ *True* and $r \leftrightarrow$ *False*.

Stålmarck's Method: Implementation

These rules are called *triggers*.

0-saturation is done by using the triggers to deduce new equivalences until nothing new can be obtained or a contradiction ($True \leftrightarrow False$) is derived.

Stålmarck's Method: Implementation

The overall algorithm works as follows:

1. The formula is negated and converted to triplets.
2. 0-saturation is performed. If a contradiction is obtained, we are done.
3. Otherwise, 1-saturation is performed: for each variable, the dilemma rule is used with $R = S_0$ to deduce new equivalences. If a contradiction is obtained, we are done.
4. Continue performing additional levels of saturation until a contradiction is obtained.

Note that the algorithm as given does not detect satisfiable formulas, only unsatisfiable formulas.

With some modification, the algorithm can be adapted to detect satisfiability as well.

Stålmarck's Method: Performance

The procedure is quite effective in many cases.

For primality formulas, it is generally comparable to DPLL. For Ramsey formulas, significantly worse. But for adder formulas it is substantially better.

Another class of formulas on which Stålmarck performs well is the so-called *urquhart* formulas:

$$p_1 \leftrightarrow p_2 \leftrightarrow \cdots \leftrightarrow p_n \leftrightarrow p_1 \leftrightarrow p_2 \leftrightarrow \cdots \leftrightarrow p_n.$$

These formulas are all 2-easy, whereas DPLL must search through nearly all possible cases to prove them.

In general, if a formula with m connectives is n-easy, Stålmarck's Method can decide it in time $O(m^{2n+1})$.

Abstract DPLL

We now return to DPLL. To facilitate a deeper look at DPLL, we use a high-level framework called *Abstract DPLL* [NOT06].

Abstract DPLL

We now return to DPLL. To facilitate a deeper look at DPLL, we use a high-level framework called *Abstract DPLL* [NOT06].

- Abstract DPLL uses *states* and *transitions* to model the progress of the algorithm.

Abstract DPLL

We now return to DPLL. To facilitate a deeper look at DPLL, we use a high-level framework called *Abstract DPLL* [NOT06].

- Abstract DPLL uses *states* and *transitions* to model the progress of the algorithm.
- Most states are of the form $M \parallel F$, where
  - M is a *sequence of* annotated *literals* denoting a partial truth assignment, and
  - F is the CNF formula being checked, represented as a *set of clauses*.

Abstract DPLL

We now return to DPLL. To facilitate a deeper look at DPLL, we use a high-level framework called *Abstract DPLL* [NOT06].

- Abstract DPLL uses *states* and *transitions* to model the progress of the algorithm.
- Most states are of the form $M \parallel F$, where
  - M is a *sequence of* annotated *literals* denoting a partial truth assignment, and
  - F is the CNF formula being checked, represented as a *set of clauses*.
- The *initial state* is $\emptyset \parallel F$, where F is to be checked for satisfiability.

Abstract DPLL

We now return to DPLL. To facilitate a deeper look at DPLL, we use a high-level framework called *Abstract DPLL* [NOT06].

- Abstract DPLL uses *states* and *transitions* to model the progress of the algorithm.
- Most states are of the form $M \parallel F$, where
  - M is a *sequence of* annotated *literals* denoting a partial truth assignment, and
  - F is the CNF formula being checked, represented as a *set of clauses*.
- The *initial state* is $\emptyset \parallel F$, where F is to be checked for satisfiability.
- Transitions between states are defined by a set of *conditional transition rules*.

Abstract DPLL

The *final state* is either:

- a special fail state: $fail$, if F is unsatisfiable, or
- $M \parallel G$, where G is a CNF formula equisatisfiable with the original formula F, and M satisfies G

We write $M \models C$ to mean that for every truth assignment v, $v(M) =$ *True* implies $v(C) =$ *True*.

Abstract DPLL Rules

UnitProp :

$$M \parallel F, C \vee l \implies M\ l \parallel F, C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \text{ is undefined in } M \end{cases}$$

PureLiteral :

$$M \parallel F \implies M\ l \parallel F \quad \textbf{if} \begin{cases} l \text{ occurs in some clause of } F \\ -l \text{ occurs in no clause of } F \\ l \text{ is undefined in } M \end{cases}$$

Decide :

$$M \parallel F \implies M\ l^{\mathsf{d}} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$$

Backtrack :

$$M\ l^{\mathsf{d}}\ N \parallel F, C \implies M\ \neg l \parallel F, C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C \\ N \text{ contains no decision literals} \end{cases}$$

Fail :

$$M \parallel F, C \implies \mathit{fail} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \text{ contains no decision literals} \end{cases}$$

Example

$$\emptyset \parallel \quad 1 \vee \overline{2},\ \ \overline{1} \vee \overline{2},\ \ 2 \vee 3,\ \ \overline{3} \vee 2,\ \ 1 \vee 4 \quad \Longrightarrow \quad \text{(PureLiteral)}$$

$$4\ 1^{d}\ \overline{2}\ 3 \parallel$$

Example

$$\begin{array}{rcllllllc}
\emptyset & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{PureLiteral}) \\
4 & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & & \\
4\ 1^{\mathrm{d}}\ \overline{2}\ 3 & \| & & & & & & &
\end{array}$$

Example

$$\begin{array}{rcl}
\emptyset & \| & 1\vee\overline{2},\ \ \overline{1}\vee\overline{2},\ \ 2\vee 3,\ \ \overline{3}\vee 2,\ \ 1\vee 4 \quad \Longrightarrow \quad (\textsf{PureLiteral}) \\
4 & \| & 1\vee\overline{2},\ \ \overline{1}\vee\overline{2},\ \ 2\vee 3,\ \ \overline{3}\vee 2,\ \ 1\vee 4 \quad \Longrightarrow \quad (\textsf{Decide}) \\
4\ 1^{\mathsf{d}} & \| & 1\vee\overline{2},\ \ \overline{1}\vee\overline{2},\ \ 2\vee 3,\ \ \overline{3}\vee 2,\ \ 1\vee 4 \\
4\ 1^{\mathsf{d}}\ \overline{2}\ 3 & \| &
\end{array}$$

Example

$$\begin{array}{rcl l l}
\emptyset & \| & 1\vee\overline{2},\ \ \overline{1}\vee\overline{2},\ \ 2\vee 3,\ \ \overline{3}\vee 2,\ \ 1\vee 4 & \Longrightarrow & (\textsf{PureLiteral}) \\
4 & \| & 1\vee\overline{2},\ \ \overline{1}\vee\overline{2},\ \ 2\vee 3,\ \ \overline{3}\vee 2,\ \ 1\vee 4 & \Longrightarrow & (\textsf{Decide}) \\
4\ 1^{\mathsf{d}} & \| & 1\vee\overline{2},\ \ \overline{1}\vee\overline{2},\ \ 2\vee 3,\ \ \overline{3}\vee 2,\ \ 1\vee 4 & \Longrightarrow & (\textsf{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2} & \| & 1\vee\overline{2},\ \ \overline{1}\vee\overline{2},\ \ 2\vee 3,\ \ \overline{3}\vee 2,\ \ 1\vee 4 & & \\
4\ 1^{\mathsf{d}}\ \overline{2}\ 3 & \| & & &
\end{array}$$

Example

$$
\begin{array}{rclcl}
\emptyset & \| & 1\lor\overline{2},\ \ \overline{1}\lor\overline{2},\ \ 2\lor 3,\ \ \overline{3}\lor 2,\ \ 1\lor 4 & \Longrightarrow & (\mathsf{PureLiteral}) \\
4 & \| & 1\lor\overline{2},\ \ \overline{1}\lor\overline{2},\ \ 2\lor 3,\ \ \overline{3}\lor 2,\ \ 1\lor 4 & \Longrightarrow & (\mathsf{Decide}) \\
4\ 1^{\mathsf{d}} & \| & 1\lor\overline{2},\ \ \overline{1}\lor\overline{2},\ \ 2\lor 3,\ \ \overline{3}\lor 2,\ \ 1\lor 4 & \Longrightarrow & (\mathsf{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2} & \| & 1\lor\overline{2},\ \ \overline{1}\lor\overline{2},\ \ 2\lor 3,\ \ \overline{3}\lor 2,\ \ 1\lor 4 & \Longrightarrow & (\mathsf{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2}\ 3 & \| & 1\lor\overline{2},\ \ \overline{1}\lor\overline{2},\ \ 2\lor 3,\ \ \overline{3}\lor 2,\ \ 1\lor 4 & &
\end{array}
$$

Example

$$
\begin{array}{rcllllllcl}
\emptyset & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & & (\text{PureLiteral}) \\
4 & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & & (\text{Decide}) \\
4\ 1^{\mathsf{d}} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & & (\text{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & & (\text{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2}\ 3 & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & & (\text{Backtrack}) \\
4\ \overline{1} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & & &
\end{array}
$$

Example

$$
\begin{array}{rcl}
\emptyset \parallel 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{3}\vee 2,\ 1\vee 4 & \Longrightarrow & (\text{PureLiteral}) \\
4 \parallel 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{3}\vee 2,\ 1\vee 4 & \Longrightarrow & (\text{Decide}) \\
4\ 1^{\mathsf{d}} \parallel 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{3}\vee 2,\ 1\vee 4 & \Longrightarrow & (\text{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2} \parallel 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{3}\vee 2,\ 1\vee 4 & \Longrightarrow & (\text{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2}\ 3 \parallel 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{3}\vee 2,\ 1\vee 4 & \Longrightarrow & (\text{Backtrack}) \\
4\ \overline{1} \parallel 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{3}\vee 2,\ 1\vee 4 & \Longrightarrow & (\text{UnitProp}) \\
4\ \overline{1}\ \overline{2}\ \overline{3} \parallel 1\vee\overline{2},\ \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{3}\vee 2,\ 1\vee 4 & &
\end{array}
$$

Example

$$
\begin{array}{rcllllllc}
\emptyset & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{PureLiteral}) \\
4 & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{Decide}) \\
4\ 1^{\mathsf{d}} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2}\ 3 & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{Backtrack}) \\
4\ \overline{1} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{UnitProp}) \\
4\ \overline{1}\ \overline{2}\ \overline{3} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{Fail}) \\
 & \mathit{fail} & & & & & & &
\end{array}
$$

Example

$$
\begin{array}{rcllllllll}
\emptyset & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{PureLiteral}) \\
4 & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{Decide}) \\
4\ 1^{\mathsf{d}} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{UnitProp}) \\
4\ 1^{\mathsf{d}}\ \overline{2}\ 3 & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{Backtrack}) \\
4\ \overline{1} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{UnitProp}) \\
4\ \overline{1}\ \overline{2}\ \overline{3} & \| & 1\vee\overline{2}, & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{3}\vee 2, & 1\vee 4 & \Longrightarrow & (\text{Fail}) \\
 & \mathit{fail} & & & & & & &
\end{array}
$$

Result: *Unsatisfiable*

Abstract DPLL: Backjumping and Learning

The basic rules can be improved by replacing the Backtrack rule with the more powerful Backjump rule and adding a Learn rule:

Backjump :

$$M\ l^{\mathsf{d}}\ N \parallel F, C \quad\Longrightarrow\quad M\ l' \parallel F, C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N \end{cases}$$

Learn :

$$M \parallel F \quad\Longrightarrow\quad M \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } F \\ F \models C \end{cases}$$

Abstract DPLL: Backjumping and Learning

The Backjump rule is best understood by introducing the notion of *implication graph*, a directed graph associated with a state $M \parallel F$ of Abstract DPLL:

- The vertices are the *variables* in M
- There is an edge from v_1 to v_2 if v_2 was assigned a value as the result of an application of UnitProp using a clause containing v_2.

When we reach a state in which $M \models \neg C$ for some $C \in F$, we add an extra *conflict* vertex and edges from each of the variables in C to the conflict vertex.

Abstract DPLL: Backjumping and Learning

The clause to use for backjumping (called the *conflict clause*) is obtained from the resulting graph:

- We first cut the graph along edges in such a way that it separates the conflict vertex from all of the decision vertices.
- Then, every vertex with an outgoing edge that was cut is marked.
- For each literal l in M whose variable is marked, $-l$ is added to the conflict clause.

To avoid ever having the same conflict again, we can learn the conflict clause using the $learn$ rule.

Example

$$\emptyset \;\|\; \overline{1}\vee 2,\;\; 3\vee 4,\;\; \overline{5}\vee\overline{6},\;\; \overline{2}\vee\overline{5}\vee 6$$

$$1^{\mathrm{d}}\; 2\; 3^{\mathrm{d}}\; 5^{\mathrm{d}}\; \overline{6} \;\|$$

Example

$$
\begin{array}{rcll}
\emptyset & \| & \overline{1} \vee 2,\;\; 3 \vee 4,\;\; \overline{5} \vee \overline{6},\;\; \overline{2} \vee \overline{5} \vee 6 & \Longrightarrow \quad (\text{Decide}) \\
1^{\mathsf{d}} & \| & \overline{1} \vee 2,\;\; 3 \vee 4,\;\; \overline{5} \vee \overline{6},\;\; \overline{2} \vee \overline{5} \vee 6 & \\
1^{\mathsf{d}}\; 2\; 3^{\mathsf{d}}\; 5^{\mathsf{d}}\; \overline{6} & \| & &
\end{array}
$$

Example

$$\begin{array}{rcllll}
\emptyset & \| & \overline{1} \vee 2, & 3 \vee 4, & \overline{5} \vee \overline{6}, & \overline{2} \vee \overline{5} \vee 6 & \Longrightarrow & (\mathsf{Decide}) \\
1^{\mathsf{d}} & \| & \overline{1} \vee 2, & 3 \vee 4, & \overline{5} \vee \overline{6}, & \overline{2} \vee \overline{5} \vee 6 & \Longrightarrow & (\mathsf{UnitProp}) \\
1^{\mathsf{d}}\ 2 & \| & \overline{1} \vee 2, & 3 \vee 4, & \overline{5} \vee \overline{6}, & \overline{2} \vee \overline{5} \vee 6 & & \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} & \| & & & & & &
\end{array}$$

Example

$$
\begin{array}{rcllll}
\emptyset & \| & \overline{1} \vee 2, & 3 \vee 4, & \overline{5} \vee \overline{6}, & \overline{2} \vee \overline{5} \vee 6 & \Longrightarrow & (\textsf{Decide}) \\
1^{\mathsf{d}} & \| & \overline{1} \vee 2, & 3 \vee 4, & \overline{5} \vee \overline{6}, & \overline{2} \vee \overline{5} \vee 6 & \Longrightarrow & (\textsf{UnitProp}) \\
1^{\mathsf{d}}\ 2 & \| & \overline{1} \vee 2, & 3 \vee 4, & \overline{5} \vee \overline{6}, & \overline{2} \vee \overline{5} \vee 6 & \Longrightarrow & (\textsf{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} & \| & \overline{1} \vee 2, & 3 \vee 4, & \overline{5} \vee \overline{6}, & \overline{2} \vee \overline{5} \vee 6 & & \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} & \| & & & & & &
\end{array}
$$

Example

$$\begin{array}{rcllllcl}
\emptyset & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Decide}) \\
1^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{UnitProp}) \\
1^{\mathsf{d}}\ 2 & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & & \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} & \| & & & & & &
\end{array}$$

Example

$$
\begin{array}{rcllll}
\emptyset & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\mathsf{Decide}) \\
1^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\mathsf{UnitProp}) \\
1^{\mathsf{d}}\ 2 & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\mathsf{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\mathsf{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\mathsf{UnitProp}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & &
\end{array}
$$

Example

$$
\begin{array}{rcllllcl}
\emptyset & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Decide}) \\
1^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{UnitProp}) \\
1^{\mathsf{d}}\ 2 & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{UnitProp}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Learn}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6, & \overline{2}\vee\overline{5} &
\end{array}
$$

Example

$$
\begin{array}{rcllllcl}
\emptyset & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\textsf{Decide}) \\
1^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\textsf{UnitProp}) \\
1^{\mathsf{d}}\ 2 & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\textsf{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\textsf{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\textsf{UnitProp}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} & \| & \overline{1}\vee 2, & 3\vee 4, & \overline{5}\vee\overline{6}, & \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\textsf{Learn})
\end{array}
$$

$$
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} \ \|\ \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6,\ \overline{2}\vee\overline{5}
$$

$$
\Longrightarrow \quad (\textsf{Backjump})
$$

$$
1^{\mathsf{d}}\ 2\ \overline{5} \ \|\ \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6,\ \overline{2}\vee\overline{5}
$$

Example

$$
\begin{array}{rcl}
\emptyset \;\|\; \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Decide}) \\
1^{\mathsf{d}} \;\|\; \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{UnitProp}) \\
1^{\mathsf{d}}\ 2 \;\|\; \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} \;\|\; \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}} \;\|\; \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{UnitProp}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} \;\|\; \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6 & \Longrightarrow & (\text{Learn})
\end{array}
$$

$$1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} \;\|\; \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6,\ \overline{2}\vee\overline{5}$$

$$\Longrightarrow \quad (\text{Backjump})$$

$$1^{\mathsf{d}}\ 2\ \overline{5} \;\|\; \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6,\ \overline{2}\vee\overline{5}$$

$$\Longrightarrow \quad (\text{Decide})$$

$$1^{\mathsf{d}}\ 2\ \overline{5}\ 3^{\mathsf{d}} \;\|\; \overline{1}\vee 2,\ 3\vee 4,\ \overline{5}\vee\overline{6},\ \overline{2}\vee\overline{5}\vee 6,\ \overline{2}\vee\overline{5}$$

Example

$$
\begin{array}{rcllllcl}
\emptyset & \| & \overline{1}\lor 2, & 3\lor 4, & \overline{5}\lor\overline{6}, & \overline{2}\lor\overline{5}\lor 6 & \Longrightarrow & (\textsf{Decide}) \\
1^{\mathsf{d}} & \| & \overline{1}\lor 2, & 3\lor 4, & \overline{5}\lor\overline{6}, & \overline{2}\lor\overline{5}\lor 6 & \Longrightarrow & (\textsf{UnitProp}) \\
1^{\mathsf{d}}\ 2 & \| & \overline{1}\lor 2, & 3\lor 4, & \overline{5}\lor\overline{6}, & \overline{2}\lor\overline{5}\lor 6 & \Longrightarrow & (\textsf{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} & \| & \overline{1}\lor 2, & 3\lor 4, & \overline{5}\lor\overline{6}, & \overline{2}\lor\overline{5}\lor 6 & \Longrightarrow & (\textsf{Decide}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}} & \| & \overline{1}\lor 2, & 3\lor 4, & \overline{5}\lor\overline{6}, & \overline{2}\lor\overline{5}\lor 6 & \Longrightarrow & (\textsf{UnitProp}) \\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} & \| & \overline{1}\lor 2, & 3\lor 4, & \overline{5}\lor\overline{6}, & \overline{2}\lor\overline{5}\lor 6 & \Longrightarrow & (\textsf{Learn})
\end{array}
$$

$$
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 5^{\mathsf{d}}\ \overline{6} \ \| \ \overline{1}\lor 2,\ \ 3\lor 4,\ \ \overline{5}\lor\overline{6},\ \ \overline{2}\lor\overline{5}\lor 6,\ \ \overline{2}\lor\overline{5}
$$

$$
\Longrightarrow \quad (\textsf{Backjump})
$$

$$
1^{\mathsf{d}}\ 2\ \overline{5} \ \| \ \overline{1}\lor 2,\ \ 3\lor 4,\ \ \overline{5}\lor\overline{6},\ \ \overline{2}\lor\overline{5}\lor 6,\ \ \overline{2}\lor\overline{5}
$$

$$
\Longrightarrow \quad (\textsf{Decide})
$$

$$
1^{\mathsf{d}}\ 2\ \overline{5}\ 3^{\mathsf{d}} \ \| \ \overline{1}\lor 2,\ \ 3\lor 4,\ \ \overline{5}\lor\overline{6},\ \ \overline{2}\lor\overline{5}\lor 6,\ \ \overline{2}\lor\overline{5}
$$

Result: *Satisfiable*

Abstract DPLL Modulo Theories Rules

Two final rules also have to do with learning:

- If too many clauses are learned, performance suffers. It is useful to *forget* some clauses (typically those that have not participated in an application of UnitProp for a while).
- If we are stuck, we can *restart* by throwing away M. Since we have learned clauses, this means our efforts were not entirely wasted. Randomly restarting can improve performance dramatically.

Forget :

$$M \parallel F, C \quad \Longrightarrow \quad M \parallel F \quad \textbf{if} \left\{ \; F \models C \right.$$

Restart :

$$M \parallel F \quad \Longrightarrow \quad \emptyset \parallel F$$

Decision Heuristics

The rules do not give any strategy for *how* to pick a variable when applying Decide.

In practice, this is critical for performance.

There are many heuristics, but the most successful currently use very cheap heuristics to try to prefer variables that are frequently involved in conflicts.

Boolean Constraint Propagation

The most expensive part of a SAT solver is the part that checks for and applies instances of the UnitProp rule.

A key insight that can be used to speed this up is that as long as a clause has at least two unassigned literals, it cannot participate in an application of UnitProp.

For every clause, we assign two of its unassigned literals as the *watched* literals.

Every time a literal is assigned, only those clauses in which it is watched need to be checked for a possible triggering of the UnitProp rule.

For those clauses that are inspected, if UnitProp is not triggered, a new unassigned literal is chosen to be watched.

Other Considerations

Modern SAT solvers [ES03, MMZ$^+$01, MSS96, Zha97] have a number of other tricks to speed things up:

- Highly tuned code
- Optimization for cache performance
- Preprocessing and clever CNF encodings
- Automatic tuning of program parameters

What is the state-of-the-art?

D. le Berre, O. Roussel, L. Simon. “The SAT ’07 Contest”
http://www.cril.univ-artois.fr/SAT07/

SAT 2007 Competition

- 44 solvers
- 3 benchmark categories
  - Industrial
  - Crafted
  - Random

Some of the winners:

- Industrial: RSat, picosat, minisat
- Crafted: SATzilla, minisat, March-KS
- Random: SATzilla, March-KS, gnovelty+

Roadmap

Boolean Satisfiability

- Propositional Logic
- Solving SAT
- Modeling for SAT

Modeling for SAT

Modeling

- Define a finite set of possibilities called *states*.
- Model states using (vectors of) propositional variables.
- Use propositional formulas to describe legal and illegal states.
- Construct a propositional formula describing the desired state.

Solving

- Translate the formula into CNF.
- If the formula is satisfiable, the satisfying assignment gives the desired state.
- If the formula is not satisfiable, the desired state does not exist.

Example: Graph Coloring

Problems involving *graph coloring* are important in both theoretical and applied computer science.

Recall that a *graph* consists of a set V of vertices and a set E of edges, where each edge is an *unordered* pair of *distinct* vertices.

A *complete graph* on n vertices is a graph with $|V| = n$ such that E contains all possible pairs of vertices.

Example: Graph Coloring

Problems involving *graph coloring* are important in both theoretical and applied computer science.

Recall that a *graph* consists of a set V of vertices and a set E of edges, where each edge is an *unordered* pair of *distinct* vertices.

A *complete graph* on n vertices is a graph with $|V| = n$ such that E contains all possible pairs of vertices.

How many edges are in a complete graph?

Example: Graph Coloring

Problems involving *graph coloring* are important in both theoretical and applied computer science.

Recall that a *graph* consists of a set V of vertices and a set E of edges, where each edge is an *unordered* pair of *distinct* vertices.

A *complete graph* on n vertices is a graph with $|V| = n$ such that E contains all possible pairs of vertices.

How many edges are in a complete graph? $\frac{n(n-1)}{2}$

Example: Graph Coloring

Suppose we wish to color each edge of a complete graph without creating any triangles in which all the edges have the same color.

What is the largest complete graph for which this is possible? The answer depends on the number of colors we are allowed to use.

Example: Graph Coloring

Suppose we wish to color each edge of a complete graph without creating any triangles in which all the edges have the same color.

What is the largest complete graph for which this is possible? The answer depends on the number of colors we are allowed to use.

What if you are only allowed one color?

Example: Graph Coloring

Suppose we wish to color each edge of a complete graph without creating any triangles in which all the edges have the same color.

What is the largest complete graph for which this is possible? The answer depends on the number of colors we are allowed to use.

What if you are only allowed one color? **Answer:** $n = 2$

Example: Graph Coloring

Suppose we wish to color each edge of a complete graph without creating any triangles in which all the edges have the same color.

What is the largest complete graph for which this is possible? The answer depends on the number of colors we are allowed to use.

What if you are only allowed one color? **Answer:** $n = 2$

What if the number of colors is *2*?

Example: Graph Coloring

Suppose we wish to color each edge of a complete graph without creating any triangles in which all the edges have the same color.

What is the largest complete graph for which this is possible? The answer depends on the number of colors we are allowed to use.

What if you are only allowed one color? **Answer:** $n = 2$

What if the number of colors is *2*? **Answer:** $n = 5$

Example: Graph Coloring

Suppose we wish to color each edge of a complete graph without creating any triangles in which all the edges have the same color.

What is the largest complete graph for which this is possible? The answer depends on the number of colors we are allowed to use.

What if you are only allowed one color? **Answer:** $n = 2$

What if the number of colors is *2*? **Answer:** $n = 5$

What if the number of colors is *3*?

Example: Graph Coloring

Suppose we wish to color each edge of a complete graph without creating any triangles in which all the edges have the same color.

What is the largest complete graph for which this is possible? The answer depends on the number of colors we are allowed to use.

What if you are only allowed one color? **Answer:** $n = 2$

What if the number of colors is *2*? **Answer:** $n = 5$

What if the number of colors is *3*? This is a job for *SAT*

Example: Graph Coloring

- *Define a finite set of possibilities called states.*

Example: Graph Coloring

- *Define a finite set of possibilities called states.*
  For this problem, each possible coloring is a state. There are $3^{|E|}$ possible states.

Example: Graph Coloring

- *Define a finite set of possibilities called states.*
  For this problem, each possible coloring is a state. There are $3^{|E|}$ possible states.
- *Model states using (vectors of) propositional variables.*

Example: Graph Coloring

- *Define a finite set of possibilities called states.*
  For this problem, each possible coloring is a state. There are $3^{|E|}$ possible states.
- *Model states using (vectors of) propositional variables.*
  A simple encoding uses two propositional variables for each edge. Since there are 4 possible combinations of values of two variables, this gives us a state space of $4^{|E|}$, which is larger than we need, but keeps the encoding simple.

Example: Graph Coloring

- *Define a finite set of possibilities called states.*
  For this problem, each possible coloring is a state. There are $3^{|E|}$ possible states.
- *Model states using (vectors of) propositional variables.*
  A simple encoding uses two propositional variables for each edge. Since there are 4 possible combinations of values of two variables, this gives us a state space of $4^{|E|}$, which is larger than we need, but keeps the encoding simple.
- *Use propositional formulas to describe legal and illegal states.*

Example: Graph Coloring

- *Define a finite set of possibilities called states.*
  For this problem, each possible coloring is a state. There are $3^{|E|}$ possible states.
- *Model states using (vectors of) propositional variables.*
  A simple encoding uses two propositional variables for each edge. Since there are 4 possible combinations of values of two variables, this gives us a state space of $4^{|E|}$, which is larger than we need, but keeps the encoding simple.
- *Use propositional formulas to describe legal and illegal states.*
  Since the color of each edge is modeled with 2 variables, there are 4 possible colors. We can write a set of formulas which disallow the fourth color.

Example: Graph Coloring

- *Define a finite set of possibilities called states.*
  For this problem, each possible coloring is a state. There are $3^{|E|}$ possible states.
- *Model states using (vectors of) propositional variables.*
  A simple encoding uses two propositional variables for each edge. Since there are 4 possible combinations of values of two variables, this gives us a state space of $4^{|E|}$, which is larger than we need, but keeps the encoding simple.
- *Use propositional formulas to describe legal and illegal states.*
  For example, if e_1 and e_2 are the variables for edge e, we simply require $\neg(e_1 \wedge e_2)$.

Example: Graph Coloring

- *Construct a propositional formula describing the desired state.*

Example: Graph Coloring

- *Construct a propositional formula describing the desired state.*
  The desired state is one in which there are no triangles of the same color. For each triangle made up of edges e, f, g, we require:
  $\neg((e_1 \leftrightarrow f_1) \wedge (f_1 \leftrightarrow g_1) \wedge (e_2 \leftrightarrow f_2) \wedge (f_2 \leftrightarrow g_2))$.

Example: Graph Coloring

- *Construct a propositional formula describing the desired state.*
  The desired state is one in which there are no triangles of the same color. For each triangle made up of edges e, f, g, we require:
  $\neg((e_1 \leftrightarrow f_1) \wedge (f_1 \leftrightarrow g_1) \wedge (e_2 \leftrightarrow f_2) \wedge (f_2 \leftrightarrow g_2))$.
- *Translate the formula into an equisatisfiable CNF formula.*

Example: Graph Coloring

- *Construct a propositional formula describing the desired state.*
  The desired state is one in which there are no triangles of the same color. For each triangle made up of edges e, f, g, we require:
  $\neg((e_1 \leftrightarrow f_1) \wedge (f_1 \leftrightarrow g_1) \wedge (e_2 \leftrightarrow f_2) \wedge (f_2 \leftrightarrow g_2))$.
- *Translate the formula into an equisatisfiable CNF formula.*
  This can be done using the CNF conversion algorithm described earlier.

Example: Graph Coloring

- *Construct a propositional formula describing the desired state.*
  The desired state is one in which there are no triangles of the same color. For each triangle made up of edges e, f, g, we require:
  $\neg((e_1 \leftrightarrow f_1) \wedge (f_1 \leftrightarrow g_1) \wedge (e_2 \leftrightarrow f_2) \wedge (f_2 \leftrightarrow g_2))$.
- *Translate the formula into an equisatisfiable CNF formula.*
  This can be done using the CNF conversion algorithm described earlier.
- *If the formula is satisfiable, the satisfying assignment gives the desired state.*

Example: Graph Coloring

- *Construct a propositional formula describing the desired state.*
  The desired state is one in which there are no triangles of the same color. For each triangle made up of edges e, f, g, we require:
  $\neg((e_1 \leftrightarrow f_1) \wedge (f_1 \leftrightarrow g_1) \wedge (e_2 \leftrightarrow f_2) \wedge (f_2 \leftrightarrow g_2))$.
- *Translate the formula into an equisatisfiable CNF formula.*
  This can be done using the CNF conversion algorithm described earlier.
- *If the formula is satisfiable, the satisfying assignment gives the desired state.*
  An actual coloring can be constructed by looking at the values of each variable given by the satisfying assignment.

Example: Graph Coloring

- *If the formula is not satisfiable, the desired state does not exist.*

Example: Graph Coloring

- *If the formula is not satisfiable, the desired state does not exist.*
  If the formula can be shown to be unsatisfiable, this is proof that there is no coloring.

Example: Graph Coloring

- *If the formula is not satisfiable, the desired state does not exist.*
  If the formula can be shown to be unsatisfiable, this is proof that there is no coloring.

What if the number of colors is *3*?

Example: Graph Coloring

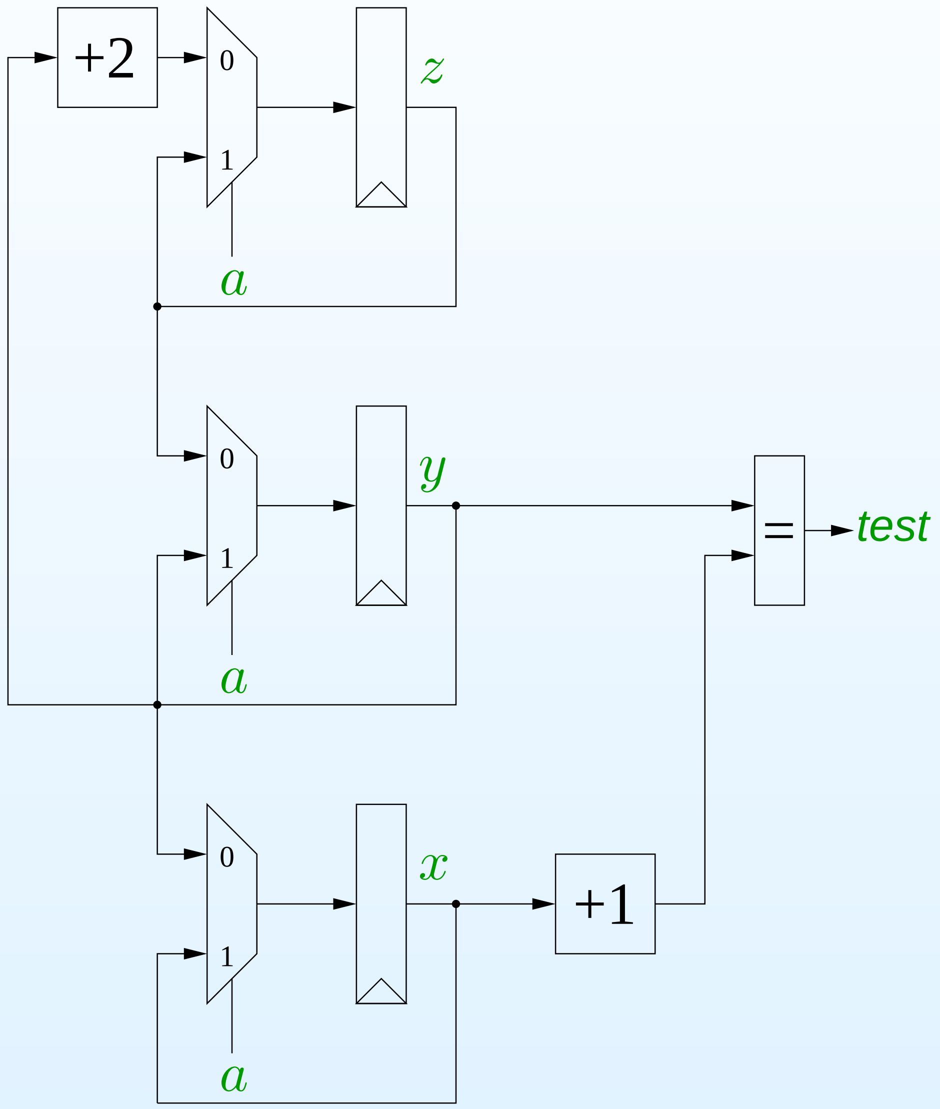
- *If the formula is not satisfiable, the desired state does not exist.*
  If the formula can be shown to be unsatisfiable, this is proof that there is no coloring.

What if the number of colors is *3*?
Answer: $n = 16$

Modeling

Let us consider again the circuit example we saw before.

Circuit Example



Modeling

One way to prove the property of the circuit is by induction.

The inductive step is essentially the following:

```
(y = x + 1 AND z = x + 2 AND
 x′ = IF a THEN x ELSE y AND
 y′ = IF a THEN y ELSE z AND
 z′ = IF a THEN z ELSE y + 2) IMPLIES
 y′ = x′ + 1 AND z′ = x′ + 2
```

We can prove this formula by showing that the negation is unsatisfiable.

We can write this formula in propositional logic by using one propositional variable for each bit in the current and next states.

Modeling

Assuming a bit-width of 2 for simplicity and skipping the details, we get the following formula:

$$
\begin{aligned}
&(z1 \leftrightarrow \neg x1) \wedge (z0 \leftrightarrow x0) \wedge \\
&(y1 \leftrightarrow (x1 \oplus x0)) \wedge (y0 \leftrightarrow \neg x0) \wedge \\
&(a \longrightarrow ((xp1 \leftrightarrow x1) \wedge (xp0 \leftrightarrow x0))) \wedge \\
&(\neg a \longrightarrow ((xp1 \leftrightarrow y1) \wedge (xp0 \leftrightarrow y0))) \wedge \\
&(a \longrightarrow ((yp1 \leftrightarrow y1) \wedge (yp0 \leftrightarrow y0))) \wedge \\
&(\neg a \longrightarrow ((yp1 \leftrightarrow z1) \wedge (yp0 \leftrightarrow z0))) \wedge \\
&(a \longrightarrow ((zp1 \leftrightarrow z1) \wedge (zp0 \leftrightarrow z0))) \wedge \\
&(\neg a \longrightarrow ((zp1 \leftrightarrow \neg y1) \wedge (zp0 \leftrightarrow y0))) \wedge \\
&(\neg(zp1 \leftrightarrow \neg xp1) \vee \neg(zp0 \leftrightarrow xp0) \vee \\
&\neg(yp1 \leftrightarrow (xp1 \oplus xp0)) \wedge (yp0 \leftrightarrow \neg xp0)
\end{aligned}
$$

Modeling: Transition Systems

Often, we want to model a system as a *transition system*: a system with a set of states and a set of possible transitions between states.

Suppose Q is a set of states, $Q_0 \subseteq Q$ a set of initial states, and T a transition relation on states (i.e. $T \subseteq Q \times Q$).

Since Q is finite, we can find an m such that $2^m \geq |Q|$. We can then use m variables: $\mathbf{x} = [x_1, \ldots, x_m]$ to represent the states. These are called *state variables*.

To represent T, we need m additional variables, $\mathbf{y} = [y_1, \ldots, y_m]$, which we call *next-state variables*.

We can write formulas $F_{Q_0}(\mathbf{x})$ and $F_T(\mathbf{y})$ such that the solutions of $F_{Q_0}(\mathbf{x})$ correspond to initial states in Q_0 and the solutions of $F_T(\mathbf{x}, \mathbf{y})$ correspond to valid transitions in T.

Bounded Model Checking

Bounded Model Checking [BCCZ99, CBRZ01] can be used to determine whether a state is reachable from the initial state in some bounded number of transitions.

To perofrm bounded model checking to a depth of n using SAT, we need n extra copies of the state variables and a set of states Q_P that we are trying to reach.

Let $\mathbf{x_0}, \ldots, \mathbf{x_n}$ be $n + 1$ copies of the state variables. And let $F_{Q_P}(\mathbf{x})$ be a formula that is true for the states in Q_P.

Q_P is reachable in n steps iff the following formula is satisfiable:

$$F_{Q_0}(\mathbf{x_0}) \wedge F_T(\mathbf{x_0}, \mathbf{x_1}) \wedge \cdots \wedge F_T(\mathbf{x_{n-1}}, \mathbf{x}) \wedge F_{Q_P}(\mathbf{x_n}).$$

Exercise

You have probably seen the following puzzle before. There is a triangle of 15 pegs with one missing. You have to jump pegs until there is only one left.

```
        X
      X   X
    X   O   X
  X   X   X   X
X   X   X   X   X
```

Exercise

Can you solve this puzzle?

Exercise

Can you solve this puzzle?

Can you solve this puzzle using SAT?

Exercise

Can you solve this puzzle?

Can you solve this puzzle using SAT?

Code for graph coloring problem is at
http://www.cs.nyu.edu/∼barrett/tmp/colors.tar

Exercise

Can you solve this puzzle?

Can you solve this puzzle using SAT?

Code for graph coloring problem is at
http://www.cs.nyu.edu/∼barrett/tmp/colors.tar

Solution on Friday...

References

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1573 of *LNCS*, pages 193–207. Springer-Verlag, 1999

[CBRZ01] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Formal Methods in System Design*, 19(1):7–34, 2001

[Coo71] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962

[End00] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Undergraduate Texts in Mathematics. Academic Press, second edition edition, 2000

[ES03] Niklas Een and Niklas Sörensson. An extensible sat-solver. In *SAT '03*, 2003

References

[Gat02] Bill Gates. Keynote address at WinHec 2002, April 2002

[Har09] John Harrison. *Introduction to Logic and Automated Theorem Proving*. 2009. Unpublished, used with permission

[KP94] W. Kunz and D. K. Pradhan. Recursive learning: A new implication technique for efficient solutions to CAD problems–test, verification, and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(9):1143–1157, September 1994

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001

[MSS96] Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006

References

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 440–446, 1992

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. J Windley, editors, *Proceedings of the 2nd International Conference on Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *LNCS*, pages 82–99. Springer, November 1998

[Tse70] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, part II*, pages 115–125, 1970

[Zha97] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 272–275. Springer, July 1997